Implementation of the Sthreads library

```
#ifndef STHREADS_H
#define STHREADS_H

#ifndef _WIN32
#error ERROR: Win32 sthreads.h included in non-Win32 program.
#endif

#ifndef _MT
#error ERROR: Sthreads program must be linked with multithreaded libraries.
#endif

#ifdef __cplusplus
extern "C" {
#endif
/*------------------------------------------------------------------------*/
/* Sthreads: A Structured Thread Library for Shared-Memory Multiprocessing  */
/* Version 1.0 for Windows NT                                               */
/*                                                                          */
/* Author: John Thornley, Computer Science Dept., Caltech.                 */
/* Date: September 1998.                                                    */
/*                                                                          */
/* Copyright (c) 1998 by John Thornley.                                     */
/*------------------------------------------------------------------------*/

/*------------------------------------------------------------------------*/
/* Error codes                                                              */
/*------------------------------------------------------------------------*/

#define STHREADS_ERROR_NONE            0
#define STHREADS_ERROR_INPUTVALUE      1
#define STHREADS_ERROR_MEMORYALLOC     2
#define STHREADS_ERROR_THREADCREATE    3
#define STHREADS_ERROR_SYNCCREATE      4
#define STHREADS_ERROR_INITIALIZED     5
#define STHREADS_ERROR_UNINITIALIZED   6
#define STHREADS_ERROR_FINALIZED       7
#define STHREADS_ERROR_INUSE           8
#define STHREADS_ERROR_LOCKHELD        9
#define STHREADS_ERROR_LOCKNOTHELD     10
#define STHREADS_ERROR_COUNTEROVERFLOW 11
#define STHREADS_ERROR_UNSPECIFIED     12

/* Requirements:                                                            */
/* - STHREADS_ERROR_NONE == 0.                                              */
/* - STHREADS_ERROR_INPUTVALUE      > STHREADS_ERROR_NONE.                  */
/* - STHREADS_ERROR_MEMORYALLOC     > STHREADS_ERROR_INPUTVALUE.            */
/* - STHREADS_ERROR_THREADCREATE    > STHREADS_ERROR_MEMORYALLOC.           */
/* - STHREADS_ERROR_SYNCCREATE      > STHREADS_ERROR_THREADCREATE.          */
/* - STHREADS_ERROR_INITIALIZED     > STHREADS_ERROR_SYNCCREATE.            */
/* - STHREADS_ERROR_UNINITIALIZED   > STHREADS_ERROR_INITIALIZED.           */
/* - STHREADS_ERROR_FINALIZED       > STHREADS_ERROR_UNINITIALIZED.         */
/* - STHREADS_ERROR_INUSE           > STHREADS_ERROR_FINALIZED.             */
/* - STHREADS_ERROR_LOCKHELD        > STHREADS_ERROR_INUSE.                 */
/* - STHREADS_ERROR_LOCKNOTHELD     > STHREADS_ERROR_LOCKHELD.              */
/* - STHREADS_ERROR_COUNTEROVERFLOW > STHREADS_ERROR_LOCKNOTHELD.           */
/* - STHREADS_ERROR_UNSPECIFIED     > STHREADS_ERROR_COUNTEROVERFLOW.       */
/* - STHREADS_ERROR_UNSPECIFIED < INT_MAX.                                  */

/*------------------------------------------------------------------------*/
/* Error string maximum length                                              */
/*------------------------------------------------------------------------*/

#define STHREADS_ERROR_STRING_MAX 100

/* Requirements:                                                            */
/* - STHREADS_ERROR_STRING_MAX >= 1.                                        */
/* - STHREADS_ERROR_STRING_MAX <= INT_MAX.                                  */

/*------------------------------------------------------------------------*/
/* Processors                                                               */
/*------------------------------------------------------------------------*/

#define STHREADS_PROCESSORS_MAX 32
```

```
#define STHREADS_PROCESSOR_YES 1000
#define STHREADS_PROCESSOR_NO  1001

/* Requirements:                                                             */
/* - STHREADS_PROCESSORS_MAX >= 1.                                           */
/* - STHREADS_PROCESSORS_MAX <= INT_MAX.                                     */
/* - STHREADS_PROCESSOR_YES >= INT_MIN.                                      */
/* - STHREADS_PROCESSOR_YES <= INT_MAX.                                      */
/* - STHREADS_PROCESSOR_NO >= INT_MIN.                                       */
/* - STHREADS_PROCESSOR_NO <= INT_MAX.                                       */
/* - STHREADS_PROCESSOR_YES != STHREADS_PROCESSOR_NO.                        */

/* Definitions:                                                              */
/* - ValidProcessorStatus(p) =                                               */
/*       p == STHREADS_PROCESSOR_PRESENT ||                                  */
/*       p == STHREADS_PROCESSOR_NOT_PRESENT.                                */

/*---------------------------------------------------------------------------*/
/* Mappings of statements/iterations to threads                              */
/*---------------------------------------------------------------------------*/

#define STHREADS_MAPPING_SIMPLE      3000
#define STHREADS_MAPPING_DYNAMIC     3001
#define STHREADS_MAPPING_BLOCKED     3002
#define STHREADS_MAPPING_INTERLEAVED 3003

/* Requirements:                                                             */
/* - STHREADS_MAPPING_SIMPLE > 0.                                            */
/* - STHREADS_MAPPING_DYNAMIC == STHREADS_MAPPING_SIMPLE + 1.                */
/* - STHREADS_MAPPING_BLOCKED == STHREADS_MAPPING_DYNAMIC + 1.               */
/* - STHREADS_MAPPING_INTERLEAVED == STHREADS_MAPPING_BLOCKED + 1.           */
/* - STHREADS_MAPPING_INTERLEAVED < INT_MAX.                                 */

/* Definitions:                                                              */
/* - ValidMapping(m) =                                                       */
/*       m == STHREADS_MAPPING_SIMPLE ||                                     */
/*       m == STHREADS_MAPPING_DYNAMIC ||                                    */
/*       m == STHREADS_MAPPING_BLOCKED ||                                    */
/*       m == STHREADS_MAPPING_INTERLEAVED.                                  */

/*---------------------------------------------------------------------------*/
/* Conditions testable in regular for loop control                           */
/*---------------------------------------------------------------------------*/

#define STHREADS_CONDITION_LT 4000
#define STHREADS_CONDITION_LE 4001
#define STHREADS_CONDITION_GT 4002
#define STHREADS_CONDITION_GE 4003

/* Requirements:                                                             */
/* - STHREADS_CONDITION_LT > 0.                                              */
/* - STHREADS_CONDITION_LE == STHREADS_CONDITION_LT + 1.                     */
/* - STHREADS_CONDITION_GT == STHREADS_CONDITION_LE + 1.                     */
/* - STHREADS_CONDITION_GE == STHREADS_CONDITION_GT + 1.                     */
/* - STHREADS_CONDITION_GE < INT_MAX.                                        */

/* Definitions:                                                              */
/* - ValidCondition(c) =                                                     */
/*       c == STHREADS_CONDITION_LT ||                                       */
/*       c == STHREADS_CONDITION_LE ||                                       */
/*       c == STHREADS_CONDITION_GT ||                                       */
/*       c == STHREADS_CONDITION_GE.                                         */

/*---------------------------------------------------------------------------*/
/* Stack sizes (in bytes)                                                    */
/*---------------------------------------------------------------------------*/

#define STHREADS_STACK_SIZE_MINIMUM  16384
#define STHREADS_STACK_SIZE_DEFAULT 262144

/* Requirements:                                                             */
/* - STHREADS_STACK_SIZE_MINIMUM >= 0.                                       */
/* - STHREADS_STACK_SIZE_DEFAULT >= STHREADS_STACK_SIZE_MINIMUM.             */
/* - STHREADS_STACK_SIZE_DEFAULT <= UINT_MAX.                                */
/* Definitions:                                                               */
/* - ValidStackSize(s) =                                                      */
/*       s >= STHREADS_STACK_SIZE_MINIMUM.                                    */

/*----------------------------------------------------------------------------*/
/* Priorities                                                                 */
/*----------------------------------------------------------------------------*/

#define STHREADS_PRIORITY_LOWEST  -2
#define STHREADS_PRIORITY_HIGHEST +2
#define STHREADS_PRIORITY_PARENT  10000 /* Inherit priority of parent thread. */

/* Requirements:                                                              */
/* - STHREADS_PRIORITY_LOWEST > INT_MIN.                                      */
/* - STHREADS_PRIORITY_HIGHEST >= STHREADS_PRIORITY_LOWEST.                   */
/* - STHREADS_PRIORITY_HIGHEST < INT_MAX.                                     */
/* - STHREADS_PRIORITY_PARENT < STHREADS_PRIORITY_LOWEST ||                   */
/*   STHREADS_PRIORITY_PARENT > STHREADS_PRIORITY_HIGHEST.                    */

/* Definitions:                                                               */
/* - ValidPriority(p) =                                                       */
/*       STHREADS_PRIORITY_LOWEST <= p && p <= STHREADS_PRIORITY_HIGHEST.     */

/*----------------------------------------------------------------------------*/
/* Print error message to string                                              */
/*----------------------------------------------------------------------------*/

void SthreadsWriteErrorMessage(int errorCode, char errorString[]);

/* Input Arguments:                                                           */
/* - errorCode : error code returned by an Sthreads function call.           */
/* Output Arguments:                                                          */
/* - errorString : error message as a char string.                            */
/* Preconditions:                                                             */
/* - errorString != NULL &&                                                   */
/*   errorString is a string of at least STHREADS_ERROR_STRING_MAX chars.     */
/* Postconditions:                                                            */
/* - errorString is '\0' terminated string of chars in the range ' ' .. '~'.  */
/* - 1 <= strlen(errorString) < STHREADS_ERROR_STRING_MAX.                    */
/* Atomicity:                                                                 */
/* - Atomic with respect to all operations.                                   */

/*----------------------------------------------------------------------------*/
/* Handle errors.                                                             */
/*----------------------------------------------------------------------------*/

void SthreadsErrorHandler(int errorCode);

/* Input Arguments:                                                           */
/* - errorCode : error code returned by an Sthreads function call.           */
/* Operation:                                                                 */
/* - error handler function is called with errorCode as argument.             */
/* Default Error Handler Function:                                            */
/* - Displays error message and terminates normal program execution.          */
/* Atomicity:                                                                 */
/* - Not atomic with respect to SthreadsSetErrorHandler operations.           */
/* - Atomic with respect to all other operations.                             */

/*----------------------------------------------------------------------------*/
/* Set error handler function.                                                */
/*----------------------------------------------------------------------------*/

int SthreadsSetErrorHandler(void (*errorHandler)(int errorCode));

/* Input Arguments:                                                           */
/* - errorHandler : function to handle errors.                                */
/* Preconditions:                                                             */
/* - errorHandler == NULL ||                                                  */
/*   errorHandler is valid void (*)(int) function.                            */
/* Postconditions:                                                            */
/* - if (errorHandler == NULL)                                                */
/*       error handler function is set to default error handler function.     */
/* - if (errorHandler != NULL)                                                */
/*       error handler func     is set to ErrorHandler.                       */
/* Atomicity:                                                                 */
/* - Not atomic with respect to                                               */
/*   SthreadsHandleError and SthreadsSetErrorHandler operations.              */
/* - Atomic with respect to all other operations.                             */

/*----------------------------------------------------------------------------*/
/* Control the processors used by program execution.                          */
/*----------------------------------------------------------------------------*/

int SthreadsGetSystemProcessors(int processor[]);

/* Output Arguments:                                                          */
/* - processors : processors that exist on the system.                        */
/* Function Return:                                                           */
/* - error code.                                                              */
/* Preconditions:                                                             */
/* - processor != NULL &&                                                     */
/*   processor is an array of at least STHREADS_PROCESSORS_MAX ints.         */
/* Postconditions:                                                            */
/* - forall (p = 0; p < STHREADS_PROCESSORS_MAX; p++)                         */
/*       ValidProcessorStatus(processor[p]) &&                                */
/*       (if (processor[p] == STHREADS_PROCESSOR_YES)                         */
/*           a processor numbered p exists on the system) &&                  */
/*       (if (processor[p] == STHREADS_PROCESSOR_NO)                          */
/*           a processor numbered p does not exist on the system).            */
/* Atomicity:                                                                 */
/* - Atomic with respect to all operations.                                   */

int SthreadsSetProgramProcessors(int processor[]);

/* Input Arguments:                                                           */
/* - processor : processors on which the threads of the program may execute. */
/* Function Return:                                                           */
/* - error code.                                                              */
/* Preconditions:                                                             */
/* - processor != NULL &&                                                     */
/*   processor is an array of at least STHREADS_PROCESSORS_MAX ints.         */
/* - forall (p = 0; p < STHREADS_PROCESSORS_MAX; p++)                         */
/*       ValidProcessorStatus(processor[p]) &&                                */
/*       if (processor[p] == STHREADS_PROCESSOR_YES)                          */
/*           a processor numbered p exists on the system.                     */
/* - exists (p = 0; p < STHREADS_PROCESSORS_MAX; p++)                         */
/*       processor[p] == STHREADS_PROCESSOR_YES.                              */
/* Atomicity:                                                                 */
/* - Must be called when program execution consists of a single thread.      */

int SthreadsGetProgramProcessors(int processor[]);

/* Output Arguments:                                                          */
/* - processors : processors on which the program may execute.                */
/* Function Return:                                                           */
/* - error code.                                                              */
/* Preconditions:                                                             */
/* - processor != NULL &&                                                     */
/*   processor is an array of at least STHREADS_PROCESSORS_MAX ints.         */
/* Postconditions:                                                            */
/* - forall (p = 0; p < STHREADS_PROCESSORS_MAX; p++)                         */
/*       ValidProcessorStatus(processor[p]) &&                                */
/*       (if (processor[p] == STHREADS_PROCESSOR_YES)                         */
/*           the program may execute on processor number p) &&                */
/*       (if (processor[p] == STHREADS_PROCESSOR_NO)                          */
/*           the program may not execute on processor number p).              */
/* Atomicity:                                                                 */
/* - Not atomic with respect to                                               */
/*   SetProgramProcessors and SetNumProgramProcessors operations.             */
/* - Atomic with respect to all other operations.                             */

int SthreadsSetThreadProcessors(int processor[]);

/* Input Arguments:                                                           */
/* - processor : processors on which the thread may execute.                  */
/* Function Return:                                                           */
/* - error code.                                                              */
/* Preconditions:                                                             */
/* - processor != NULL &&                                                     */
/*   processor is an array of at least STHREADS_PROCESSORS_MAX ints.          */
/* - forall (p = 0; p < STHREADS_PROCESSORS_MAX; p++)                         */
/*       ValidProcessorStatus(processor[p]) &&                                */
/*       if (processor[p] == STHREADS_PROCESSOR_YES)                          */
/*           the program may execute on processor number p.                   */
/* - exists (p = 0; p < STHREADS_PROCESSORS_MAX; p++)                         */
/*       processor[p] == STHREADS_PROCESSOR_YES.                              */
/* Atomicity:                                                                 */
/* - Not atomic with respect to                                               */
/*   SetProgramProcessors and SetNumProgramProcessors operations.             */
/* - Atomic with respect to all other operations.                             */

int SthreadsGetNumSystemProcessors(int *numProcessors);

/* Output Arguments:                                                          */
/* - numProcessors : number of processors that exist on the system.           */
/* Function Return:                                                           */
/* - error code.                                                              */
/* Preconditions:                                                             */
/* - numProcessors != NULL && numProcessors points to a valid int variable.   */
/* Postconditions:                                                            */
/* - *numProcessors == number of processors that exist on the system.         */
/* Atomicity:                                                                 */
/* - Atomic with respect to all operations.                                   */

int SthreadsSetNumProgramProcessors(int numProcessors);

/* Input Arguments:                                                           */
/* - numProcessors : number of processors on which the threads of the program */
/*                   may execute.                                             */
/* Function Return:                                                           */
/* - error code.                                                              */
/* Preconditions:                                                             */
/* - numProcessors >= 1.                                                      */
/* - numProcessors <= number of processors that exist on the system.          */
/* Atomicity:                                                                 */
/* - Must be called when program execution consists of a single thread.       */

/*----------------------------------------------------------------------------*/
/* Multithreaded block                                                        */
/*----------------------------------------------------------------------------*/

int SthreadsBlock(
       int numStatements, void (*statement[])(void *args), void *args,
       int mapping, int numThreads,
       int priority, unsigned int stackSize);

/* Input Arguments:                                                           */
/* - numStatements : number of statements in block.                           */
/* - statement     : functions representing statements.                       */
/* - args          : pointer to arguments of the statements.                  */
/* - mapping       : mapping of statements onto threads.                      */
/* - numThreads    : number of threads.                                       */
/* - priority      : priority of threads.                                     */
/* - stackSize     : stack size of threads.                                   */
/* Function Return:                                                           */
/* - error code.                                                              */
/* Preconditions:                                                             */
/* - numStatements >= 0.                                                      */
/* - statement != NULL &&                                                     */
/*   statement is an array of at least numStatements functions.               */
/* - forall (s = 0; s < numStatements; s++)                                   */
/*       statement[s] != NULL &&                                              */
/*       statement[s] is a valid void (*)(void *) function.                   */
/* - ValidMapping(mapping).                                                   */
/* - if (mapping != STHREADS_MAPPING_SIMPLE)                                  */
/*       (numThreads > 0) || (numThreads == 0 && numStatements == 0).         */
/* - ValidPriority(priority) || priority == STHREADS_PRIORITY_PARENT.         */
/* - ValidStackSize(stackSize).                                               */
/* Atomicity:                                                                 */
/* - Atomic with respect to all operations.                                   */
```

```
/*----------------------------------------------------------------------------*/
/* Multithreaded regular for loop                                             */
/*----------------------------------------------------------------------------*/

int SthreadsRegularForLoop(
        void (*chunk)(int initial, int bound, int step, void *args), void *args,
        int initial, int condition, int bound, int step,
        int chunkSize, int mapping, int numThreads,
        int priority, unsigned int stackSize);

/* Input Arguments:                                                           */
/* - chunk      : function to execute iterations of loop body.                */
/* - args       : pointer to arguments of loop body.                          */
/* - initial    : initial value of control variable.                          */
/* - condition  : condition between control variable and bound value.         */
/* - bound      : bound value of control variable.                            */
/* - step       : step value of control variable.                             */
/* - chunkSize  : number of iterations per chunk.                             */
/* - mapping    : mapping of chunks onto threads.                             */
/* - numThreads : number of threads.                                          */
/* - priority   : priority of threads.                                        */
/* - stackSize  : stack size of threads.                                      */
/* Function Return:                                                           */
/* - error code.                                                              */
/* Preconditions:                                                             */
/* -  chunk != NULL &&                                                        */
/*    chunk is a valid void (*)(int, int, int, void *) function.              */
/* - ValidCondition(condition).                                               */
/* - !InfiniteRange(initial, condition, bound, step).                         */
/* - (chunkSize > 0) ||                                                       */
/*   (chunkSize == 0 && NullRange(initial, condition, bound, step)).          */
/* - ValidMapping(mapping).                                                   */
/* - if (mapping != STHREADS_MAPPING_SIMPLE)                                  */
/*         (numThreads > 0) ||                                                */
/*         (numThreads == 0 && NullRange(initial, condition, bound, step)).   */
/* - ValidPriority(priority) || priority == STHREADS_PRIORITY_PARENT.         */
/* - ValidStackSize(stackSize).                                               */

/* Definitions:                                                               */
/* - InfiniteRange(initial, condition, bound, step) =                         */
/*         (condition == STHREADS_CONDITION_LT &&                             */
/*              initial < bound && step <= 0) ||                              */
/*         (condition == STHREADS_CONDITION_LE &&                             */
/*              initial <= bound && step <= 0) ||                             */
/*         (condition == STHREADS_CONDITION_GT &&                             */
/*              initial > bound && step >= 0) ||                              */
/*         (condition == STHREADS_CONDITION_GE &&                             */
/*              initial >= bound && step >= 0).                               */
/* - NullRange(initial, condition, bound, step) =                             */
/*         (condition == STHREADS_CONDITION_LT && initial >= bound) ||        */
/*         (condition == STHREADS_CONDITION_LE && initial > bound) ||         */
/*         (condition == STHREADS_CONDITION_GT && initial <= bound) ||        */
/*         (condition == STHREADS_CONDITION_GE && initial < bound).           */
/* Atomicity:                                                                 */
/* - Atomic with respect to all operations.                                   */

/*----------------------------------------------------------------------------*/
/* Flags                                                                      */
/*----------------------------------------------------------------------------*/

typedef struct {
    unsigned char value[16];
} SthreadsFlag;

int SthreadsFlagInitialize(SthreadsFlag *flag);

/* Input-Output Arguments:                                                    */
/* - flag : flag variable.                                                    */
/* Function Return:                                                           */
/* - error code.                                                              */
/* Preconditions:                                                             */
/* - flag != NULL && flag points to a valid flag variable.                    */
/* - !Initialized(flag).                                                      */
/* Atomicity:                                                                 */
/* - Not atomic with respec     all other operations on flag.               */
/* - Atomic with respect to      other operations.                          */

int SthreadsFlagFinalize(SthreadsFlag *flag);

/* Input-Output Arguments:                                                  */
/* - flag : flag variable.                                                  */
/* Function Return:                                                         */
/* - error code.                                                            */
/* Preconditions:                                                           */
/* - flag != NULL && flag points to a valid flag variable.                  */
/* - Initialized(flag) && !Finalized(flag).                                 */
/* - NumWaiting(flag) == 0.                                                 */
/* Atomicity:                                                               */
/* - Not atomic with respect to all other operations on flag.               */
/* - Atomic with respect to all other operations.                           */

int SthreadsFlagSet(SthreadsFlag *flag);

/* Input-Output Arguments:                                                  */
/* - flag : flag variable.                                                  */
/* Function Return:                                                         */
/* - error code.                                                            */
/* Preconditions:                                                           */
/* - flag != NULL && flag points to a valid flag variable.                  */
/* - Initialized(flag) && !Finalized(flag).                                 */
/* Atomicity:                                                               */
/* - Atomic with respect to Set and Check operations on flag.               */
/* - Not atomic with respect to other operations on flag.                   */
/* - Atomic with respect to all other operations.                           */

int SthreadsFlagCheck(SthreadsFlag *flag);

/* Input-Output Arguments:                                                  */
/* - flag : flag variable.                                                  */
/* Function Return:                                                         */
/* - error code.                                                            */
/* Preconditions:                                                           */
/* - flag != NULL && flag points to a valid flag variable.                  */
/* - Initialized(flag) && !Finalized(flag).                                 */
/* Atomicity:                                                               */
/* - Atomic with respect to Set and Check operations on flag.               */
/* - Not atomic with respect to other operations on flag.                   */
/* - Atomic with respect to all other operations.                           */

int SthreadsFlagReset(SthreadsFlag *flag);

/* Input-Output Arguments:                                                  */
/* - flag : flag variable.                                                  */
/* Function Return:                                                         */
/* - error code.                                                            */
/* Preconditions:                                                           */
/* - flag != NULL && flag points to a valid flag variable.                  */
/* - Initialized(flag) && !Finalized(flag).                                 */
/* - NumWaiting(flag) == 0.                                                 */
/* Atomicity:                                                               */
/* - Not atomic with respect to other operations on flag.                   */
/* - Atomic with respect to all other operations.                           */

/*--------------------------------------------------------------------------*/
/* Counters                                                                 */
/*--------------------------------------------------------------------------*/

typedef struct {
    unsigned char value[40];
} SthreadsCounter;

int SthreadsCounterInitialize(SthreadsCounter *counter);

/* Input-Output Arguments:                                                  */
/* - counter : pointer to counter variable.                                 */
/* Function Return:                                                         */
/* - error code.                                                            */
/* Preconditions:                                                           */
/* - counter != NULL && counter points to a valid counter varia           */
/* - !Initialized(counter).                                               */
/* Atomicity:                                                             */
/* - Not atomic with respect to all other operations on counter.          */
/* - Atomic with respect to all other operations.                         */

int SthreadsCounterFinalize(SthreadsCounter *counter);

/* Input-Output Arguments:                                                */
/* - counter : pointer to counter variable.                               */
/* Function Return:                                                       */
/* - error code.                                                          */
/* Preconditions:                                                         */
/* - counter != NULL && counter points to a valid counter variable.       */
/* - Initialized(counter) && !Finalized(counter).                         */
/* - NumWaiting(counter) == 0.                                            */
/* Atomicity:                                                             */
/* - Not atomic with respect to all other operations on counter.          */
/* - Atomic with respect to all other operations.                         */

int SthreadsCounterIncrement(SthreadsCounter *counter, unsigned int amount);

/* Input-Output Arguments:                                                */
/* - counter : pointer to counter variable.                               */
/* Function Return:                                                       */
/* - error code.                                                          */
/* Preconditions:                                                         */
/* - counter != NULL && counter points to a valid counter variable.       */
/* - Initialized(counter) && !Finalized(counter).                         */
/* - Count(counter) <= UINT_MAX - amount.                                 */
/* Atomicity:                                                             */
/* - Atomic with respect to Increment and Check operations on counter.    */
/* - Not atomic with respect to other operations on counter.              */
/* - Atomic with respect to all other operations.                         */

int SthreadsCounterCheck(SthreadsCounter *counter, unsigned int value);

/* Input-Output Arguments:                                                */
/* - counter : pointer to counter variable.                               */
/* Function Return:                                                       */
/* - error code.                                                          */
/* Preconditions:                                                         */
/* - counter != NULL && counter points to a valid counter variable.       */
/* - Initialized(counter) && !Finalized(counter).                         */
/* Atomicity:                                                             */
/* - Atomic with respect to Increment and Check operations on counter.    */
/* - Not atomic with respect to other operations on counter.              */
/* - Atomic with respect to all other operations.                         */

int SthreadsCounterReset(SthreadsCounter *counter);

/* Input-Output Arguments:                                                */
/* - counter : pointer to counter variable.                               */
/* Function Return:                                                       */
/* - error code.                                                          */
/* Preconditions:                                                         */
/* - counter != NULL && counter points to a valid counter variable.       */
/* - Initialized(counter) && !Finalized(counter).                         */
/* - NumWaiting(counter) == 0.                                            */
/* Atomicity:                                                             */
/* - Not atomic with respect to all other operations on counter.          */
/* - Atomic with respect to all other operations.                         */

/*------------------------------------------------------------------------*/
/* Locks                                                                  */
/*------------------------------------------------------------------------*/

typedef struct {
    unsigned char value[36];
} SthreadsLock;

int SthreadsLockInitialize(SthreadsLock *lock);

/* Input-Output Arguments:                                                */
/* - lock : pointer to lock variable.                                         */
/* Function Return:                                                           */
/* - error code.                                                              */
/* Preconditions:                                                             */
/* - lock != NULL && lock points to a valid lock variable.                    */
/* - !Initialized(lock).                                                      */
/* Atomicity:                                                                 */
/* - Not atomic with respect to all other operations on lock.                 */
/* - Atomic with respect to all other operations.                             */

int SthreadsLockFinalize(SthreadsLock *lock);

/* Input-Output Arguments:                                                    */
/* - lock : pointer to lock variable.                                         */
/* Function Return:                                                           */
/* - error code.                                                              */
/* Preconditions:                                                             */
/* - lock != NULL && lock points to a valid lock variable.                    */
/* - Initialized(lock) && !Finalized(lock).                                   */
/* - !AnyThreadHolds(lock).                                                   */
/* Atomicity:                                                                 */
/* - Not atomic with respect to all other operations on lock.                 */
/* - Atomic with respect to all other operations.                             */

int SthreadsLockAcquire(SthreadsLock *lock);

/* Input-Output Arguments:                                                    */
/* - lock : pointer to lock variable.                                         */
/* Function Return:                                                           */
/* - error code.                                                              */
/* Preconditions:                                                             */
/* - lock != NULL && lock points to a valid lock variable.                    */
/* - Initialized(lock) && !Finalized(lock).                                   */
/* - !ThisThreadHolds(lock).                                                  */
/* Atomicity:                                                                 */
/* - Atomic with respect to Acquire and Release operations on lock.           */
/* - Not atomic with respect to other operations on lock.                     */
/* - Atomic with respect to all other operations.                             */

int SthreadsLockRelease(SthreadsLock *lock);

/* Input-Output Arguments:                                                    */
/* - lock : pointer to lock variable.                                         */
/* Function Return:                                                           */
/* - error code.                                                              */
/* Preconditions:                                                             */
/* - lock != NULL && lock points to a valid lock variable.                    */
/* - Initialized(lock) && !Finalized(lock).                                   */
/* - ThisThreadHolds(lock).                                                   */
/* Atomicity:                                                                 */
/* - Atomic with respect to Acquire and Release operations on lock.           */
/* - Not atomic with respect to other operations on lock.                     */
/* - Atomic with respect to all other operations.                             */

/*----------------------------------------------------------------------------*/
/* Barriers                                                                   */
/*----------------------------------------------------------------------------*/

typedef struct {
    unsigned char value[52];
} SthreadsBarrier;

int SthreadsBarrierInitialize(SthreadsBarrier *barrier, int numThreads);

/* Input-Output Arguments:                                                    */
/* - barrier    : pointer to barrier variable.                                */
/* - numThreads : number of threads that cross barrier in each pass.          */
/* Function Return:                                                           */
/* - error code.                                                              */
/* Preconditions:                                                             */
/* - barrier != NULL && barrier points to a valid barrier variable.           */
/* - !Initialized(barrier).                                                   */
/* - numThreads >= 1.                                                         */
/* Atomicity:                                                                 */
/* - Not atomic with respec       all other operations on barrier.                 */
/* - Atomic with respect to       other operations.                                */

int SthreadsBarrierFinalize(SthreadsBarrier *barrier);

/* Input-Output Arguments:                                                         */
/* - barrier : pointer to barrier variable.                                        */
/* Function Return:                                                                */
/* - error code.                                                                   */
/* Preconditions:                                                                  */
/* - barrier != NULL && barrier points to a valid barrier variable.                */
/* - Initialized(barrier) && !Finalized(barrier).                                  */
/* - NumWaiting(barrier) == 0.                                                     */
/* Atomicity:                                                                      */
/* - Not atomic with respect to all other operations on barrier.                   */
/* - Atomic with respect to all other operations.                                  */

int SthreadsBarrierPass(SthreadsBarrier *barrier);

/* Input-Output Arguments:                                                         */
/* - barrier : pointer to barrier variable.                                        */
/* Function Return:                                                                */
/* - error code.                                                                   */
/* Preconditions:                                                                  */
/* - barrier != NULL && barrier points to a valid barrier variable.                */
/* - Initialized(barrier) && !Finalized(barrier).                                  */
/* Atomicity:                                                                      */
/* - Atomic with respect to Pass operations on barrier.                            */
/* - Not atomic with respect to other operations on barrier.                       */
/* - Atomic with respect to all other operations.                                  */

int SthreadsBarrierReset(SthreadsBarrier *barrier, int numThreads);

/* Input-Output Arguments:                                                         */
/* - barrier : pointer to barrier variable.                                        */
/* Function Return:                                                                */
/* - error code.                                                                   */
/* Preconditions:                                                                  */
/* - barrier != NULL && barrier points to a valid barrier variable.                */
/* - Initialized(barrier) && !Finalized(barrier).                                  */
/* - NumWaiting(barrier) == 0.                                                     */
/* - numThreads >= 1.                                                              */
/* Atomicity:                                                                      */
/* - Not atomic with respect to all other operations on barrier.                   */
/* - Atomic with respect to all other operations.                                  */

/*---------------------------------------------------------------------------------*/
/* Priorities                                                                      */
/*---------------------------------------------------------------------------------*/

int SthreadsGetCurrentPriority(int *priority);

/* Output Arguments:                                                               */
/* - priority : scheduling priority of calling thread.                             */
/* Function Return:                                                                */
/* - error code.                                                                   */
/* Preconditions:                                                                  */
/* - priority != NULL && priority points to a valid int variable.                  */
/* Postconditions:                                                                 */
/* - *priority == scheduling priority of calling thread.                           */
/* Atomicity:                                                                      */
/* - Atomic with respect to all operations.                                        */

int SthreadsSetCurrentPriority(int priority);

/* Input Arguments:                                                                */
/* - priority : scheduling priority for calling thread.                            */
/* Function Return:                                                                */
/* - error code.                                                                   */
/* Preconditions:                                                                  */
/* - ValidPriority(priority).                                                      */
/* Atomicity:                                                                      */
/* - Atomic with respect to all operations.                                        */
```

```
/*-------------------------     ------------------------------      ---------*/
#ifdef __cplusplus
}
#endif

#endif /* !STHREADS_H */
```

```
/*---------------------------------------------------------------------------*/
/* Sthreads: A Structured Thread Library for Shared-Memory Multiprocessing    */
/* Version 1.0 for Windows NT                                                */
/*                                                                           */
/* Author: John Thornley, Computer Science Dept., Caltech.                   */
/* Date: September 1998.                                                     */
/*                                                                           */
/* Copyright (c) 1998 by John Thornley.                                      */
/*                                                                           */
/* THINGS TO DO:                                                             */
/*                                                                           */
/* - Change names of CHECK tests, e.g., to CHECKNOTINITIALIZED.              */
/* - Make Finalize operations set Initialized and Finalized flags to false.  */
/* - Counter for dynamic for loop should be unsigned int.                    */
/* - Declarations of thread functions should be compatible with              */
/*   Win32 prototype ... see page 25.                                        */
/* - Implement special case of BarrierPass when numThreads == 1.             */
/* - Implement flags like counters for efficiency when flag is set?          */
/* - Change priority low and high to THREAD_PRIORITY_IDLE and _TIME_CRITICAL. */
/*                                                                           */
/*---------------------------------------------------------------------------*/

#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <limits.h>
#include <windows.h>
#include "sthreads.h"

/*---------------------------------------------------------------------------*/
/* Bool type definition                                                      */
/*---------------------------------------------------------------------------*/

typedef int bool;
#define false 0
#define true  1

/*---------------------------------------------------------------------------*/
/* Miscellaneous utility definitions                                         */
/*---------------------------------------------------------------------------*/

#define MIN(x, y) ((x) < (y) ? (x) : (y))
#define MAX(x, y) ((x) > (y) > (x) : (y))

/*---------------------------------------------------------------------------*/
/* Verify requirements, beliefs, and checks                                  */
/*---------------------------------------------------------------------------*/

#define require(condition) assert(condition) /* require this input condition  */
#define believe(condition) assert(condition) /* believe this must be true      */
#define check(condition) assert(condition)   /* check this is true             */

/*---------------------------------------------------------------------------*/
/* Check for error conditions                                                */
/*---------------------------------------------------------------------------*/

#define CHECKINPUTVALUE(condition) \
    if (!(condition)) { return STHREADS_ERROR_INPUTVALUE; }

#define CHECKMEMORYALLOC(condition) \
    if (!(condition)) { return STHREADS_ERROR_MEMORYALLOC; }

#define CHECKTHREADCREATE(condition) \
    if (!(condition)) { return STHREADS_ERROR_THREADCREATE; }

#define CHECKSYNCCREATE(condition) \
    if (!(condition)) { return STHREADS_ERROR_SYNCCREATE; }

#define CHECKINITIALIZED(condition) \
    if (!(condition)) { return STHREADS_ERROR_INITIALIZED; }

#define CHECKUNINITIALIZED(condition) \
    if (!(condition)) { return STHREADS_ERROR_UNINITIALIZED; }
```

```
#define CHECKFINALIZED(condition) \
    if (!(condition)) { return STHREADS_ERROR_FINALIZED; }

#define CHECKINUSE(condition) \
    if (!(condition)) { return STHREADS_ERROR_INUSE; }

#define CHECKLOCKHELD(condition) \
    if (!(condition)) { return STHREADS_ERROR_LOCKHELD; }

#define CHECKLOCKNOTHELD(condition) \
    if (!(condition)) { return STHREADS_ERROR_LOCKNOTHELD; }

#define CHECKCOUNTEROVERFLOW(condition) \
    if (!(condition)) { return STHREADS_ERROR_COUNTEROVERFLOW; }

#define CHECKOTHER(condition) \
    if (!(condition)) { return STHREADS_ERROR_UNSPECIFIED; }

/*-----------------------------------------------------------------------------*/
/* Is processor status value valid?                                            */
/*-----------------------------------------------------------------------------*/

static bool ValidProcessorStatus(int p)
{
    return
        p == STHREADS_PROCESSOR_YES ||
        p == STHREADS_PROCESSOR_NO;
}

/*-----------------------------------------------------------------------------*/
/* Is mapping value valid?                                                     */
/*-----------------------------------------------------------------------------*/

static bool ValidMapping(int m)
{
    return
        m == STHREADS_MAPPING_SIMPLE ||
        m == STHREADS_MAPPING_DYNAMIC ||
        m == STHREADS_MAPPING_BLOCKED ||
        m == STHREADS_MAPPING_INTERLEAVED;
}

/*-----------------------------------------------------------------------------*/
/* Is condition value valid?                                                   */
/*-----------------------------------------------------------------------------*/

static bool ValidCondition(int c)
{
    return
        c == STHREADS_CONDITION_LT ||
        c == STHREADS_CONDITION_LE ||
        c == STHREADS_CONDITION_GT ||
        c == STHREADS_CONDITION_GE;
}

/*-----------------------------------------------------------------------------*/
/* Is stack-size value valid?                                                  */
/*-----------------------------------------------------------------------------*/

static bool ValidStackSize(unsigned int s)
{
    return
        s >= STHREADS_STACK_SIZE_MINIMUM;
}

/*-----------------------------------------------------------------------------*/
/* Is priority value valid?                                                    */
/*-----------------------------------------------------------------------------*/

static bool ValidPriority(int p)
{
    return
        STHREADS_PRIORITY_LOWEST <= p && p <= STHREADS_PRIORITY_HIGHEST;
```

```
}

/*----------------------------------------------------------------------------*/
/* Print error message to string                                              */
/*----------------------------------------------------------------------------*/

void SthreadsWriteErrorMessage(int errorCode, char errorString[])
{
    switch (errorCode) {
    case STHREADS_ERROR_NONE:
        sprintf(errorString,
                "no error");
        break;
    case STHREADS_ERROR_INPUTVALUE:
        sprintf(errorString,
                "input value precondition violation");
        break;
    case STHREADS_ERROR_MEMORYALLOC:
        sprintf(errorString,
                "memory allocation failure");
        break;
    case STHREADS_ERROR_THREADCREATE:
        sprintf(errorString,
                "system thread creation failure");
        break;
    case STHREADS_ERROR_SYNCCREATE:
        sprintf(errorString,
                "system synchronization creation failure");
        break;
    case STHREADS_ERROR_INITIALIZED:
        sprintf(errorString,
                "initialization on previously initialized object");
        break;
    case STHREADS_ERROR_UNINITIALIZED:
        sprintf(errorString,
                "operation on uninitialized object");
        break;
    case STHREADS_ERROR_FINALIZED:
        sprintf(errorString,
                "operation on finalized object");
        break;
    case STHREADS_ERROR_INUSE:
        sprintf(errorString,
                "finalization/reset on in-use object");
        break;
    case STHREADS_ERROR_LOCKNOTHELD:
        sprintf(errorString,
                "release on lock not held");
        break;
    case STHREADS_ERROR_COUNTEROVERFLOW:
        sprintf(errorString,
                "counter overflow");
        break;
    case STHREADS_ERROR_UNSPECIFIED:
        sprintf(errorString,
                "unspecified error");
        break;
    default:
        sprintf(errorString,
                ">>>>> unknown error code <<<<<");
        break;
    }
}

/*----------------------------------------------------------------------------*/
/* Default error handler function:                                            */
/* displays error message and terminate  normal program execution.            */
/*----------------------------------------------------------------------------*/

static void DefaultErrorHandler(int errorCode)
{
    char errorString[STHREADS_ERROR_STRING_MAX];

    if (errorCode != STHREADS_ERROR_NONE) {
        SthreadsWriteErrorMessage(errorCode, errorString);
        fprintf(stderr, "\n%s\n", errorString);
        exit(EXIT_FAILURE);
    }
}

/*------------------------------------------------------------------------------*/
/* Error handler function.                                                      */
/*------------------------------------------------------------------------------*/

static void (*errorHandlerFunction)(int errorCode) = DefaultErrorHandler;

/*------------------------------------------------------------------------------*/
/* Handle errors.                                                               */
/*------------------------------------------------------------------------------*/

#define UNLOCKED 0
#define LOCKED   1

static LONG lock = UNLOCKED;

void SthreadsErrorHandler(int errorCode)
{
    while (InterlockedExchange((LPLONG) &lock, LOCKED) != UNLOCKED);
    (*errorHandlerFunction)(errorCode);
    InterlockedExchange((LPLONG) &lock, UNLOCKED);
}

#undef UNLOCKED
#undef LOCKED

/*------------------------------------------------------------------------------*/
/* Set error handler function.                                                  */
/*------------------------------------------------------------------------------*/

int SthreadsSetErrorHandler(void (*errorHandler)(int errorCode))
{
    if (errorHandler == NULL)
        errorHandlerFunction = DefaultErrorHandler;
    else
        errorHandlerFunction = errorHandler;

    return STHREADS_ERROR_NONE;
}

/*------------------------------------------------------------------------------*/
/* Control the processors used by program execution.                            */
/*------------------------------------------------------------------------------*/

int SthreadsGetSystemProcessors(int processor[])
{
    DWORD processAffinity, systemAffinity, processorBit;
    int p;

    require(STHREADS_PROCESSORS_MAX == 32);
    GetProcessAffinityMask(
        GetCurrentProcess(),
        (LPDWORD) &processAffinity, (LPDWORD) &systemAffinity);

    CHECKINPUTVALUE(processor != NULL);

    processorBit = (DWORD) 1;
    for (p = 0; p < STHREADS_PROCESSORS_MAX; p++) {
        if (systemAffinity & processorBit)
            processor[p] = STHREADS_PROCESSOR_YES;
        else
            processor[p] = STHREADS_PROCESSOR_NO;
        processorBit = processorBit << 1;
    }

    return STHREADS_ERROR_NONE;
}

/*------------------------------------------------------------------------------*/
```

```
int SthreadsSetProgramProcessors(int processor[])
{
    DWORD processAffinity, systemAffinity, processorBit;
    int p;

    require(STHREADS_PROCESSORS_MAX == 32);
    GetProcessAffinityMask(
        GetCurrentProcess(),
        (LPDWORD) &processAffinity, (LPDWORD) &systemAffinity);

    CHECKINPUTVALUE(processor != NULL);
    processorBit = (DWORD) 1;
    for (p = 0; p < STHREADS_PROCESSORS_MAX; p++) {
        CHECKINPUTVALUE(ValidProcessorStatus(processor[p]));
        if (processor[p] == STHREADS_PROCESSOR_YES)
            CHECKINPUTVALUE(systemAffinity & processorBit);
        processorBit = processorBit << 1;
    }
    for (p = 0; p < STHREADS_PROCESSORS_MAX; p++)
        if (processor[p] == STHREADS_PROCESSOR_YES) break;
    CHECKINPUTVALUE(p < STHREADS_PROCESSORS_MAX);

    processAffinity = (DWORD) 0;
    processorBit = (DWORD) 1;
    for (p = 0; p < STHREADS_PROCESSORS_MAX; p++) {
        if (processor[p] == STHREADS_PROCESSOR_YES)
            processAffinity = processAffinity | processorBit;
        processorBit = processorBit << 1;
    }
    SetProcessAffinityMask(GetCurrentProcess(), processAffinity);
    SetThreadAffinityMask(GetCurrentThread(), processAffinity);

    return STHREADS_ERROR_NONE;
}

/*----------------------------------------------------------------------------*/

int SthreadsGetProgramProcessors(int processor[])
{
    DWORD processAffinity, systemAffinity, processorBit;
    int p;

    require(STHREADS_PROCESSORS_MAX == 32);
    GetProcessAffinityMask(
        GetCurrentProcess(),
        (LPDWORD) &processAffinity, (LPDWORD) &systemAffinity);

    CHECKINPUTVALUE(processor != NULL);

    processorBit = (DWORD) 1;
    for (p = 0; p < STHREADS_PROCESSORS_MAX; p++) {
        if (processAffinity & processorBit)
            processor[p] = STHREADS_PROCESSOR_YES;
        else
            processor[p] = STHREADS_PROCESSOR_NO;
        processorBit = processorBit << 1;
    }

    return STHREADS_ERROR_NONE;
}

/*----------------------------------------------------------------------------*/

int SthreadsSetThreadProcessors(int processor[])
{
    DWORD threadAffinity, processAffinity, systemAffinity, processorBit;
    int p;

    require(STHREADS_PROCESSORS_MAX == 32);
    GetProcessAffinityMask(
        GetCurrentProcess(),
        (LPDWORD) &processAffinity, (LPDWORD) &systemAffinity);
    CHECKINPUTVALUE(process[illegible]= NULL);
    processorBit = (DWORD) 1;
    for (p = 0; p < STHREADS_PROCESSORS_MAX; p++) {
        CHECKINPUTVALUE(ValidProcessorStatus(processor[p]));
        if (processor[p] == STHREADS_PROCESSOR_YES)
            CHECKINPUTVALUE(processAffinity & processorBit);
        processorBit = processorBit << 1;
    }
    for (p = 0; p < STHREADS_PROCESSORS_MAX; p++)
        if (processor[p] == STHREADS_PROCESSOR_YES) break;
    CHECKINPUTVALUE(p < STHREADS_PROCESSORS_MAX);

    threadAffinity = (DWORD) 0;
    processorBit = (DWORD) 1;
    for (p = 0; p < STHREADS_PROCESSORS_MAX; p++) {
        if (processor[p] == STHREADS_PROCESSOR_YES)
            threadAffinity = threadAffinity | processorBit;
       processorBit = processorBit << 1;
    }
    SetThreadAffinityMask(GetCurrentThread(), threadAffinity);

    return STHREADS_ERROR_NONE;
}

/*----------------------------------------------------------------------------*/

int SthreadsGetNumSystemProcessors(int *numProcessors)
{
    DWORD processAffinity, systemAffinity, processorBit;
    int p, count;

    require(STHREADS_PROCESSORS_MAX == 32);
    GetProcessAffinityMask(
        GetCurrentProcess(),
        (LPDWORD) &processAffinity, (LPDWORD) &systemAffinity);

    CHECKINPUTVALUE(numProcessors != NULL);

    count = 0;
    processorBit = (DWORD) 1;
    for (p = 0; p < STHREADS_PROCESSORS_MAX; p++) {
        if (systemAffinity & processorBit)
            count = count + 1;
        processorBit = processorBit << 1;
    }
    *numProcessors = count;

    return STHREADS_ERROR_NONE;
}

/*----------------------------------------------------------------------------*/

int SthreadsSetNumProgramProcessors(int numProcessors)
{
    DWORD processAffinity, systemAffinity, processorBit;
    int p, numSystemProcessors;

    require(STHREADS_PROCESSORS_MAX == 32);
    GetProcessAffinityMask(
        GetCurrentProcess(),
        (LPDWORD) &processAffinity, (LPDWORD) &systemAffinity);

    CHECKINPUTVALUE(numProcessors >= 1);
    numSystemProcessors = 0;
    processorBit = (DWORD) 1;
    for (p = 0; p < STHREADS_PROCESSORS_MAX; p++) {
        if (systemAffinity & processorBit)
            numSystemProcessors = numSystemProcessors + 1;
        processorBit = processorBit << 1;
    }
    CHECKINPUTVALUE(numProcessors <= numSystemProcessors);

    processAffinity = (DWORD) 0;
    processorBit = (DWORD) 1;
    for (p = 0; p < STHREADS_PROCESSORS_MAX && numProcessors >        ) {
        if (systemAffinity & processorBit) {
            processAffinity = processAffinity | processorBit;
            numProcessors = numProcessors - 1;
        }
        processorBit = processorBit << 1;
    }
    believe(numProcessors == 0);
    SetProcessAffinityMask(GetCurrentProcess(), processAffinity);

    return STHREADS_ERROR_NONE;
}

/*----------------------------------------------------------------------------*/
/* Arguments for multithreaded block thread                                   */
/*----------------------------------------------------------------------------*/

typedef struct {
    int numStatements;
    void (**statement)(void *args);
    void *args;
    int first, last, step;
    int *counter;
    LPCRITICAL_SECTION counterLock;
    LPLONG threadCount;
    HANDLE threadsFinished;
} MTBargs;

/*----------------------------------------------------------------------------*/
/* Simple multithreaded block thread                                          */
/*----------------------------------------------------------------------------*/

static void SMTBthread(MTBargs *args)
{
    BOOL returnOK;

    require(args != NULL);
    require(args->numStatements > 0);
    require(args->statement != NULL);
    require(0 <= args->first && args->first < args->numStatements);
    require(*args->statement[args->first] != NULL);

    (*args->statement[args->first])(args->args);

    if (InterlockedDecrement(args->threadCount) == 0) {
        returnOK = SetEvent(args->threadsFinished);
        check(returnOK);
    }
}

/*----------------------------------------------------------------------------*/
/* Dynamic multithreaded block thread                                         */
/*----------------------------------------------------------------------------*/

static void DMTBthread(MTBargs *args)
{
    int s;
    bool finished;
    BOOL returnOK;

    require(args != NULL);
    require(args->numStatements > 0);
    require(args->statement != NULL);
    require(0 <= args->first && args->first < args->numStatements);
    require(args->counter != NULL);
    require(args->counterLock != NULL);

    s = args->first;
    while (true) {
        require(args->statement[s] != NULL);
        (*args->statement[s])(args->args);
        EnterCriticalSection(args->counterLock);
        finished = (*args->counter == args->numStatements - 1);
        if (!finished) {
            *args->counter = *args->counter + 1;
            s = *args->counter;
        }
        LeaveCriticalSection(args->counterLock);
        if (finished) break;
    }

    if (InterlockedDecrement(args->threadCount) == 0) {
        returnOK = SetEvent(args->threadsFinished);
        check(returnOK);
    }
}

/*---------------------------------------------------------------------------*/
/* Blocked and interleaved multithreaded block thread                        */
/*---------------------------------------------------------------------------*/

static void BIMTBthread(MTBargs *args)
{
    int s;
    BOOL returnOK;

    require(args != NULL);
    require(args->numStatements > 0);
    require(args->statement != NULL);
    require(0 <= args->last && args->last < args->numStatements);
    require(0 <= args->first && args->first <= args->last);
    require(args->step > 0);
    require((args->last - args->first)%args->step == 0);

    s = args->first;
    while (true) {
        require(args->statement[s] != NULL);
        (*args->statement[s])(args->args);
        if (s == args->last) break;
        believe(args->last - s >= args->step);
        s = s + args->step;
    }

    if (InterlockedDecrement(args->threadCount) == 0) {
        returnOK = SetEvent(args->threadsFinished);
        check(returnOK);
    }
}

/*---------------------------------------------------------------------------*/
/* Multithreaded block                                                       */
/*---------------------------------------------------------------------------*/

int SthreadsBlock(
       int numStatements, void (*statement[])(void *args), void *args,
       int mapping, int numThreads,
       int priority, unsigned int stackSize)
{
    HANDLE *thread;
    MTBargs *threadArgs;
    LONG threadCount;
    HANDLE threadsFinished;
    HANDLE parentThread;
    int parentPriority;
    void (*threadStart)(MTBargs *args);
    int s, t;
    DWORD threadID;
    int counter;
    CRITICAL_SECTION counterLock;
    int blockFirst, blockSize, blockRemainder;
    BOOL returnOK;
    DWORD returnCode;

    CHECKINPUTVALUE(numStatements >= 0);
    CHECKINPUTVALUE(statement != NULL);
    for (s = 0; s < numStatements; s++)
        CHECKINPUTVALUE(statement[s] != NULL);
    CHECKINPUTVALUE(ValidMapping(mapping));
```

```
if (mapping != STHREADS_MAPPING_SIMPLE)
    CHECKINPUTVALUE((numThreads > 0) ||
               (numThreads == 0 && numStatements == 0));
CHECKINPUTVALUE(
    ValidPriority(priority) || priority == STHREADS_PRIORITY_PARENT);
CHECKINPUTVALUE(ValidStackSize(stackSize));

if (numStatements == 0) return STHREADS_ERROR_NONE;

if (mapping == STHREADS_MAPPING_SIMPLE) numThreads = numStatements;
if (numThreads > numStatements) numThreads = numStatements;
if (numThreads == 1) mapping = STHREADS_MAPPING_BLOCKED;
if (numThreads == numStatements) mapping = STHREADS_MAPPING_SIMPLE;

CHECKMEMORYALLOC(numThreads <= INT_MAX/sizeof(HANDLE));
thread = (HANDLE *) malloc(numThreads*sizeof(HANDLE));
CHECKMEMORYALLOC(thread != NULL);
CHECKMEMORYALLOC(numThreads <= INT_MAX/sizeof(MTBargs));
threadArgs = (MTBargs *) malloc(numThreads*sizeof(MTBargs));
CHECKMEMORYALLOC(threadArgs != NULL);

parentThread = GetCurrentThread();
believe(parentThread != NULL);
parentPriority = GetThreadPriority(parentThread);
believe(parentPriority != THREAD_PRIORITY_ERROR_RETURN);
believe(ValidPriority(parentPriority));
if (priority != STHREADS_PRIORITY_PARENT) {
    returnOK = SetThreadPriority(parentThread, priority);
    believe(returnOK);
}

switch (mapping) {
case STHREADS_MAPPING_SIMPLE:
    threadStart = SMTBthread;
    break;
case STHREADS_MAPPING_DYNAMIC:
    counter = numThreads - 1;
    InitializeCriticalSection(&counterLock);
    threadStart = DMTBthread;
    break;
case STHREADS_MAPPING_BLOCKED:
    blockFirst = 0;
    blockSize = numStatements/numThreads;
    blockRemainder = numStatements%numThreads;
    threadStart = BIMTBthread;
    break;
case STHREADS_MAPPING_INTERLEAVED:
    blockSize = numStatements/numThreads;
    blockRemainder = numStatements%numThreads;
    threadStart = BIMTBthread;
    break;
default:
    assert(false);
}

threadCount = numThreads;
threadsFinished = CreateEvent(NULL, TRUE, FALSE, NULL);
CHECKSYNCCREATE(threadsFinished != NULL);
for (t = 0; t < numThreads; t++) {
    threadArgs[t].numStatements = numStatements;
    threadArgs[t].statement = statement;
    threadArgs[t].args = args;
    threadArgs[t].threadCount = (LPLONG) &threadCount;
    threadArgs[t].threadsFinished = threadsFinished;

    switch (mapping) {
    case STHREADS_MAPPING_SIMPLE:
        threadArgs[t].first = t;
        break;
    case STHREADS_MAPPING_DYNAMIC:
        threadArgs[t].first = t;
        threadArgs[t].counter = &counter;
        threadArgs[t].counterLock = &counterLock;
        break;
        case STHREADS_MAPPING_BLOCKED:
            threadArgs[t].first = blockFirst;
            threadArgs[t].last = blockFirst + (blockSize - 1);
            threadArgs[t].step = 1;
            if (blockRemainder > 0) {
                threadArgs[t].last = threadArgs[t].last + 1;
                blockRemainder = blockRemainder - 1;
            }
            blockFirst = threadArgs[t].last + 1;
            break;
        case STHREADS_MAPPING_INTERLEAVED:
            threadArgs[t].first = t;
            threadArgs[t].last = blockSize*numThreads + t;
            threadArgs[t].step = numThreads;
            if (blockRemainder == 0)
                threadArgs[t].last = threadArgs[t].last - numThreads;
            else
                blockRemainder = blockRemainder - 1;
            break;
        default:
            believe(false);
        }

        thread[t] = CreateThread(NULL, stackSize,
            (LPTHREAD_START_ROUTINE) threadStart,
            (LPVOID) &threadArgs[t], CREATE_SUSPENDED, &threadID);
        CHECKTHREADCREATE(thread[t] != NULL);
        if (priority == STHREADS_PRIORITY_PARENT)
            returnOK = SetThreadPriority(thread[t], parentPriority);
        else
            returnOK = SetThreadPriority(thread[t], priority);
        CHECKTHREADCREATE(returnOK);
        returnCode = ResumeThread(thread[t]);
        CHECKTHREADCREATE(returnCode == 1);
    }

    if (priority != STHREADS_PRIORITY_PARENT) {
        returnOK = SetThreadPriority(parentThread, parentPriority);
        believe(returnOK);
    }
    returnCode = WaitForSingleObject(threadsFinished, INFINITE);
    CHECKOTHER(returnCode != WAIT_FAILED);
    returnOK = CloseHandle(threadsFinished);
    CHECKOTHER(returnOK == TRUE);
    for (t = 0; t < numThreads; t++) {
        returnOK = CloseHandle(thread[t]);
        CHECKOTHER(returnOK == TRUE);
    }
    if (mapping == STHREADS_MAPPING_DYNAMIC)
        DeleteCriticalSection(&counterLock);
    free(thread);
    free(threadArgs);

    return STHREADS_ERROR_NONE;
}

/*----------------------------------------------------------------------------*/
/* Is regular for loop range infinite?                                        */
/*----------------------------------------------------------------------------*/

static bool InfiniteRange(int initial, int condition, int bound, int step)
{
    require(ValidCondition(condition));

    switch (condition) {
    case STHREADS_CONDITION_LT:
        return initial < bound && step <= 0;
    case STHREADS_CONDITION_LE:
        return initial <= bound && step <= 0;
    case STHREADS_CONDITION_GT:
        return initial > bound && step >= 0;
    case STHREADS_CONDITION_GE:
        return initial >= bound && step >= 0;
    default:
        believe(false);
        return false; /* This return should never be executed. */
    }
}

/*-------------------------------------------------------------------------*/
/* Is regular for loop range null?                                         */
/*-------------------------------------------------------------------------*/

static bool NullRange(int initial, int condition, int bound, int step)
{
    require(ValidCondition(condition));

    switch (condition) {
    case STHREADS_CONDITION_LT:
        return initial >= bound;
    case STHREADS_CONDITION_LE:
        return initial > bound;
    case STHREADS_CONDITION_GT:
        return initial <= bound;
    case STHREADS_CONDITION_GE:
        return initial < bound;
    default:
        believe(false);
        return false; /* This return should never be executed. */
    }
}

/*-------------------------------------------------------------------------*/
/* Arithmetic operations on signed and unsigned integers                   */
/*-------------------------------------------------------------------------*/

static unsigned int DIFF(int high, int low)
{
    require(low <= high);

    return (unsigned int) (high - low);
}

/*-------------------------------------------------------------------------*/

static int ADD(int base, unsigned int offset)
{
    require(offset <= DIFF(INT_MAX, base));

    return base + (int) offset;
}

/*-------------------------------------------------------------------------*/

static int SUBTRACT(int base, unsigned int offset)
{
    require(offset <= DIFF(base, INT_MIN));

    return base - (int) offset;
}

/*-------------------------------------------------------------------------*/
/* Split range 0 .. rangeLast into chunks numbered 0 .. chunkLast with     */
/* chunks.  Return the first and last indices of chunk c.                  */
/*-------------------------------------------------------------------------*/

static void SPLIT(
        unsigned int rangeLast, unsigned int chunkLast, unsigned int c,
        unsigned int *first, unsigned int *last)
{
    unsigned int smallerChunkSize;
    unsigned int numLargerChunks;

    require(chunkLast <= rangeLast);
    require(c <= chunkLast);
    require(first != NULL && last != NULL);

    if (chunkLast == 0) {
        *first = 0;
        *last = rangeLast;
    } else if (chunkLast == rangeLast) {
        *first = c;
        *last = c;
    } else {
        smallerChunkSize = (rangeLast - chunkLast)/(chunkLast + 1) + 1;
        numLargerChunks = (rangeLast - chunkLast)%(chunkLast + 1);
        *first = c*smallerChunkSize + MIN(c, numLargerChunks);
        *last = *first + (smallerChunkSize - 1);
        if (c < numLargerChunks) *last = *last + 1;
    }
}

/*----------------------------------------------------------------------------*/
/* Last iteration number in regular for loop range                            */
/* (iterations numbered 0, 1, 2, ...)                                         */
/*----------------------------------------------------------------------------*/

static unsigned int LAST_ITERATION_NUM(
        int initial, int condition, int bound, int step)
{
    require(ValidCondition(condition));
    require(!InfiniteRange(initial, condition, bound, step));
    require(!NullRange(initial, condition, bound, step));

    switch (condition) {
    case STHREADS_CONDITION_LT:
        believe(initial < bound && step > 0);
        return DIFF(bound - 1, initial)/((unsigned int) step);
    case STHREADS_CONDITION_LE:
        believe(initial <= bound && step > 0);
        return DIFF(bound, initial)/((unsigned int) step);
    case STHREADS_CONDITION_GT:
        believe(initial > bound && step < 0);
        return DIFF(initial, bound + 1)/((unsigned int) -step);
    case STHREADS_CONDITION_GE:
        believe(initial >= bound && step < 0);
        return DIFF(initial, bound)/((unsigned int) -step);
    default:
        assert(false);
        return false; /* This return should never be executed. */
    }
}

/*----------------------------------------------------------------------------*/
/* Last chunk number in regular for loop range (chunks numbered 0, 1, 2, ...) */
/*----------------------------------------------------------------------------*/

static unsigned int LAST_CHUNK_NUM(
        int initial, int condition, int bound, int step, int chunkSize)
{
    require(ValidCondition(condition));
    require(!InfiniteRange(initial, condition, bound, step));
    require(!NullRange(initial, condition, bound, step));
    require(chunkSize >= 1);

    return LAST_ITERATION_NUM(initial, condition, bound, step)/
           ((unsigned int) chunkSize);
}

/*----------------------------------------------------------------------------*/
/* Control value on ith iteration of regular for loop range (i = 0, 1, 2, ...)*/
/*----------------------------------------------------------------------------*/

static int ControlValue(unsigned int i, int initial, int step)
{
    require(step != 0);

    if (step > 0)
        return ADD(initial, i*((unsigned int) step));
    else
        return SUBTRACT(initial, i*((unsigned int) -step));
}
```

```
/*----------------------------------------------------------------------------*/
/* Does control value lie inside regular for loop range?                      */
/*----------------------------------------------------------------------------*/

static bool InRange(
        int controlValue, int initial, int condition, int bound, int step)
{
    require(ValidCondition(condition));
    require(!InfiniteRange(initial, condition, bound, step));
    require(!NullRange(initial, condition, bound, step));

    switch (condition) {
    case STHREADS_CONDITION_LT:
        believe(step > 0);
        return initial <= controlValue && controlValue < bound;
    case STHREADS_CONDITION_LE:
        believe(step > 0);
        return initial <= controlValue && controlValue <= bound;
    case STHREADS_CONDITION_GT:
        believe(step < 0);
        return initial >= controlValue && controlValue > bound;
    case STHREADS_CONDITION_GE:
        believe(step < 0);
        return initial >= controlValue && controlValue >= bound;
    default:
        believe(false);
        return false; /* This return should never be executed. */
    }
}

/*----------------------------------------------------------------------------*/
/* Execute cth chunk of regular for loop range  (c = 0, 1, 2, ...)            */
/*----------------------------------------------------------------------------*/

static void ExecuteChunk(
        int initial, int condition, int bound, int step, int chunkSize,
        unsigned int c, void (*chunk)(int, int, int, void *), void *args)
{
    unsigned int iFirst, iLast;
    int chunkInitial, chunkLast, chunkBound;

    require(ValidCondition(condition));
    require(!InfiniteRange(initial, condition, bound, step));
    require(!NullRange(initial, condition, bound, step));
    require(chunkSize >= 1);
    require(c <= LAST_CHUNK_NUM(initial, condition, bound, step, chunkSize));
    require(chunk != NULL);

    SPLIT(
        LAST_ITERATION_NUM(initial, condition, bound, step),
        LAST_CHUNK_NUM(initial, condition, bound, step, chunkSize), c,
        &iFirst, &iLast);
    believe(0 <= iFirst);
    believe(iFirst <= iLast);
    believe(iLast <= LAST_ITERATION_NUM(initial, condition, bound, step));
    chunkInitial = ControlValue(iFirst, initial, step);
    believe(InRange(chunkInitial, initial, condition, bound, step));
    chunkLast = ControlValue(iLast, initial, step);
    believe(InRange(chunkLast, initial, condition, bound, step));
    switch (condition) {
    case STHREADS_CONDITION_LT:
        chunkBound = chunkLast + 1;
        break;
    case STHREADS_CONDITION_LE:
        chunkBound = chunkLast;
        break;
    case STHREADS_CONDITION_GT:
        chunkBound = chunkLast - 1;
        break;
    case STHREADS_CONDITION_GE:
        chunkBound = chunkLast;
        break;
    default:
        believe(false);
    }
    (*chunk)(chunkInitial, chunkBound, step, args);
}

/*----------------------------------------------------------------------------*/
/* Arguments for multithreaded regular for loop thread                        */
/*----------------------------------------------------------------------------*/

typedef struct {
    void (*chunk)(int initial, int bound, int step, void *args);
    void *args;
    int initial, condition, bound, step;
    int chunkSize;
    unsigned int chunkFirst, chunkLast, chunkStep;
    unsigned int *counter;
    LPCRITICAL_SECTION counterLock;
    LPLONG threadCount;
    HANDLE threadsFinished;
} MTRFLargs;

/*----------------------------------------------------------------------------*/
/* Simple multithreaded regular for loop thread                               */
/*----------------------------------------------------------------------------*/

static void SMTRFLthread(MTRFLargs *args)
{
    BOOL returnOK;

    require(args != NULL);
    require(args->chunk != NULL);
    require(ValidCondition(args->condition));
    require(!InfiniteRange(
        args->initial, args->condition, args->bound, args->step));
    require(!NullRange(
        args->initial, args->condition, args->bound, args->step));
    require(args->chunkSize >= 1);
    require(args->chunkFirst <= LAST_CHUNK_NUM(
        args->initial, args->condition, args->bound, args->step,
        args->chunkSize));

    ExecuteChunk(
        args->initial, args->condition, args->bound, args->step,
        args->chunkSize, args->chunkFirst, args->chunk, args->args);

    if (InterlockedDecrement(args->threadCount) == 0) {
        returnOK = SetEvent(args->threadsFinished);
        check(returnOK);
    }
}

/*----------------------------------------------------------------------------*/
/* Dynamic multithreaded regular for loop thread                              */
/*----------------------------------------------------------------------------*/

static void DMTRFLthread(MTRFLargs *args)
{
    unsigned int c, last_c;
    bool finished;
    BOOL returnOK;

    require(args != NULL);
    require(args->chunk != NULL);
    require(ValidCondition(args->condition));
    require(!InfiniteRange(
        args->initial, args->condition, args->bound, args->step));
    require(!NullRange(
        args->initial, args->condition, args->bound, args->step));
    require(args->chunkSize >= 1);
    require(args->chunkFirst <= LAST_CHUNK_NUM(
        args->initial, args->condition, args->bound, args->step,
        args->chunkSize));
    require(args->counter != NULL);
    require(args->counterLock != NULL);
    c = args->chunkFirst;
    last_c = LAST_CHUNK_NUM(
        args->initial, args->condition, args->bound, args->step,
        args->chunkSize);
    while (true) {
        ExecuteChunk(
            args->initial, args->condition, args->bound, args->step,
            args->chunkSize, c, args->chunk, args->args);
        EnterCriticalSection(args->counterLock);
        finished = (*args->counter == last_c);
        if (!finished) {
            *args->counter = *args->counter + 1;
            c = *args->counter;
        }
        LeaveCriticalSection(args->counterLock);
        if (finished) break;
    }

    if (InterlockedDecrement(args->threadCount) == 0) {
        returnOK = SetEvent(args->threadsFinished);
        check(returnOK);
    }
}

/*----------------------------------------------------------------------------*/
/* Blocked and interleaved multithreaded regular for loop thread              */
/*----------------------------------------------------------------------------*/

static void BIMTRFLthread(MTRFLargs *args)
{
    unsigned int c;
    BOOL returnOK;

    require(args != NULL);
    require(args->chunk != NULL);
    require(ValidCondition(args->condition));
    require(!InfiniteRange(
        args->initial, args->condition, args->bound, args->step));
    require(!NullRange(
        args->initial, args->condition, args->bound, args->step));
    require(args->chunkSize >= 1);
    require(args->chunkFirst <= args->chunkLast);
    require(args->chunkLast <= LAST_CHUNK_NUM(
        args->initial, args->condition, args->bound, args->step,
        args->chunkSize));
    require((args->chunkLast - args->chunkFirst)%args->chunkStep == 0);

    c = args->chunkFirst;
    while (true) {
        ExecuteChunk(
            args->initial, args->condition, args->bound, args->step,
            args->chunkSize, c, args->chunk, args->args);
        if (c == args->chunkLast) break;
        believe(args->chunkLast - c >= args->chunkStep);
        c = c + args->chunkStep;
    }

    if (InterlockedDecrement(args->threadCount) == 0) {
        returnOK = SetEvent(args->threadsFinished);
        check(returnOK);
    }
}

/*----------------------------------------------------------------------------*/
/* Multithreaded regular for loop                                             */
/*----------------------------------------------------------------------------*/

int SthreadsRegularForLoop(
        void (*chunk)(int initial, int bound, int step, void *args), void *args,
        int initial, int condition, int bound, int step,
        int chunkSize, int mapping, int numThreads,
        int priority, unsigned int stackSize)
{
    unsigned int lastChunkN
    HANDLE *thread;
    MTRFLargs *threadArgs;
    LONG threadCount;
    HANDLE threadsFinished;
    HANDLE parentThread;
    int parentPriority;
    void (*thread_start)(MTRFLargs *args);
    int t;
    DWORD threadID;
    int counter;
    CRITICAL_SECTION counterLock;
    unsigned int blockFirst, blockSize, blockRemainder;
    BOOL returnOK;
    DWORD returnCode;

    CHECKINPUTVALUE(chunk != NULL);
    CHECKINPUTVALUE(ValidCondition(condition));
    CHECKINPUTVALUE(!InfiniteRange(initial, condition, bound, step));
    CHECKINPUTVALUE((chunkSize > 0) ||
               (chunkSize == 0 &&
                NullRange(initial, condition, bound, step)));
    CHECKINPUTVALUE(ValidMapping(mapping));
    if (mapping != STHREADS_MAPPING_SIMPLE)
        CHECKINPUTVALUE((numThreads > 0) ||
                   (numThreads == 0 &&
                    NullRange(initial, condition, bound, step)));
    CHECKINPUTVALUE(
        ValidPriority(priority) || priority == STHREADS_PRIORITY_PARENT);
    CHECKINPUTVALUE(ValidStackSize(stackSize));

    if (NullRange(initial, condition, bound, step))
        return STHREADS_ERROR_NONE;

    lastChunkNum = LAST_CHUNK_NUM(
        initial, condition, bound, step, chunkSize);
    CHECKMEMORYALLOC(!(mapping == STHREADS_MAPPING_SIMPLE &&
                       lastChunkNum >= INT_MAX));

    if (mapping == STHREADS_MAPPING_SIMPLE)
        numThreads = (int) (lastChunkNum + 1);
    if ((unsigned int) (numThreads - 1) > lastChunkNum)
        numThreads = (int) (lastChunkNum + 1);
    if (numThreads == 1)
        mapping = STHREADS_MAPPING_INTERLEAVED;
    if ((unsigned int) (numThreads - 1) == lastChunkNum)
        mapping = STHREADS_MAPPING_SIMPLE;

    CHECKMEMORYALLOC(numThreads <= INT_MAX/sizeof(HANDLE));
    thread = (HANDLE *) malloc(numThreads*sizeof(HANDLE));
    CHECKMEMORYALLOC(thread != NULL);
    CHECKMEMORYALLOC(numThreads <= INT_MAX/sizeof(MTRFLargs));
    threadArgs = (MTRFLargs *) malloc(numThreads*sizeof(MTRFLargs));
    CHECKMEMORYALLOC(threadArgs != NULL);

    parentThread = GetCurrentThread();
    believe(parentThread != NULL);
    parentPriority = GetThreadPriority(parentThread);
    believe(parentPriority != THREAD_PRIORITY_ERROR_RETURN);
    believe(ValidPriority(parentPriority));
    if (priority != STHREADS_PRIORITY_PARENT) {
        returnOK= SetThreadPriority(parentThread, priority);
        believe(returnOK);
    }

    switch (mapping) {
    case STHREADS_MAPPING_SIMPLE:
        thread_start = SMTRFLthread;
        break;
    case STHREADS_MAPPING_DYNAMIC:
        counter = numThreads - 1;
        InitializeCriticalSection(&counterLock);
        thread_start = DMTRFLthread;
        break;
    case STHREADS_MAPPING_BLOCKED:
        blockFirst = 0;
        blockSize =
            (lastChunkNum - (((unsigned int) numThreads) - 1))/
            ((unsigned int) numThreads) + 1;
        blockRemainder =
            (lastChunkNum - (((unsigned int) numThreads) - 1))%
            ((unsigned int) numThreads);
        thread_start = BIMTRFLthread;
        break;
    case STHREADS_MAPPING_INTERLEAVED:
        blockSize =
            (lastChunkNum - (((unsigned int) numThreads) - 1))/
            ((unsigned int) numThreads) + 1;
        blockRemainder =
            (lastChunkNum - (((unsigned int) numThreads) - 1))%
            ((unsigned int) numThreads);
        thread_start = BIMTRFLthread;
        break;
    default:
        assert(false);
    }

    threadCount = numThreads;
    threadsFinished = CreateEvent(NULL, TRUE, FALSE, NULL);
    CHECKSYNCCREATE(threadsFinished != NULL);
    for (t = 0; t < numThreads; t++) {
        threadArgs[t].chunk = chunk;
        threadArgs[t].args = args;
        threadArgs[t].initial = initial;
        threadArgs[t].condition = condition;
        threadArgs[t].bound = bound;
        threadArgs[t].step = step;
        threadArgs[t].chunkSize = chunkSize;
        threadArgs[t].threadCount = (LPLONG) &threadCount;
        threadArgs[t].threadsFinished = threadsFinished;

        switch (mapping) {
        case STHREADS_MAPPING_SIMPLE:
            threadArgs[t].chunkFirst = t;
            break;
        case STHREADS_MAPPING_DYNAMIC:
            threadArgs[t].chunkFirst = t;
            threadArgs[t].counter = &counter;
            threadArgs[t].counterLock = &counterLock;
            break;
        case STHREADS_MAPPING_BLOCKED:
            threadArgs[t].chunkFirst = blockFirst;
            threadArgs[t].chunkLast = blockFirst + (blockSize - 1);
            threadArgs[t].chunkStep = 1;
            if (blockRemainder > 0) {
                threadArgs[t].chunkLast = threadArgs[t].chunkLast + 1;
                blockRemainder = blockRemainder - 1;
            }
            blockFirst = threadArgs[t].chunkLast + 1;
            break;
        case STHREADS_MAPPING_INTERLEAVED:
            threadArgs[t].chunkFirst = t;
            threadArgs[t].chunkLast =
                blockSize*((unsigned int) numThreads) + t;
            threadArgs[t].chunkStep =
                (unsigned int) numThreads;
            if (blockRemainder == 0)
                threadArgs[t].chunkLast =
                    threadArgs[t].chunkLast - ((unsigned int) numThreads);
            else
                blockRemainder = blockRemainder - 1;
            break;
        default:
            believe(false);
        }

        thread[t] = CreateThread(NULL, stackSize,
            (LPTHREAD_START_ROUTINE) thread_start,
            (LPVOID) &threadArgs[t], CREATE_SUSPENDED, &threadI
        CHECKTHREADCREATE(thread[t] != NULL);
        if (priority == STHREADS_PRIORITY_PARENT)
            SetThreadPriority(thread[t], parentPriority);
        else
            SetThreadPriority(thread[t], priority);
        ResumeThread(thread[t]);
    }

    if (priority != STHREADS_PRIORITY_PARENT) {
        SetThreadPriority(parentThread, parentPriority);
        believe(returnOK);
    }
    returnCode = WaitForSingleObject(threadsFinished, INFINITE);
    CHECKOTHER(returnCode != WAIT_FAILED);
    returnOK = CloseHandle(threadsFinished);
    CHECKOTHER(returnOK == TRUE);
    for (t = 0; t < numThreads; t++) {
        returnOK = CloseHandle(thread[t]);
        CHECKOTHER(returnOK == TRUE);
    }
    if (mapping == STHREADS_MAPPING_DYNAMIC)
        DeleteCriticalSection(&counterLock);
    free(thread);
    free(threadArgs);

    return STHREADS_ERROR_NONE;
}

/*----------------------------------------------------------------------------*/
/* Multithreaded nested regular for loop (for future release?)               */
/*----------------------------------------------------------------------------*/

int SthreadsNestedRegularForLoop(
        int nesting,
        void (*chunk)(int first[], int last[], int step[], void *args),
        void *args,
        int initial[], int condition[], int bound[], int step[],
        int chunkSize[], int mapping[], int numThreads[],
        int priority, unsigned int stackSize)
/* Arguments:                                                                 */
/* - nesting    : degree of nesting.                                          */
/* - chunk      : function to execute chunk of iterations of loop body.       */
/* - args       : pointer to arguments of loop body.                          */
/* - initial    : initial value of control variable at each nesting level.    */
/* - condition  : condition between control variable and bound value          */
/*                at each nesting level.                                      */
/* - bound      : bound value of control variable at each nesting level.      */
/* - step       : step value of control variable at each nesting level.       */
/* - chunkSize  : number of iterations per chunk at each nesting level.       */
/* - mapping    : mapping of chunks onto threads at each nesting level.       */
/* - numThreads : number of threads at each nesting level.                    */
/* - priority   : priority of threads.                                        */
/* - stackSize  : stack size of threads.                                      */
/* Returns:                                                                   */
/* - error code.                                                              */
/* Requirements:                                                              */
/* - nesting >= 1                                                             */
/* - chunk != NULL &&                                                         */
/*   chunk is a valid void (*)(int *, int *, int *, void *) function.         */
/* - initial != NULL &&                                                       */
/*   initial is an array of at least nesting ints.                            */
/* - condition != NULL &&                                                     */
/*   condition is an array of at least nesting ints.                          */
/* - forall (i = 0; i < nesting; i++) ValidCondition(condition[i]).           */
/* - bound != NULL &&                                                         */
/*   bound is an array of at least nesting ints.                              */
/* - step != NULL &&                                                          */
/*   step is an array of at least nesting ints.                               */
/* - forall (i = 0; i < nesting; i++)                                         */
/*       !InfiniteRange(initial[i], condition[i], bound[i], step[i]) ||       */
/*       exists (j = 0; j < i; j++)                                           */
/*           NullRange(initial[j], condition[j], bound[j], step[j]).          */
/* - forall (i = 0; i < nesting; i++)                                         */
/*       (chunkSize[i] > 0) ||                                                */
/*       (chunkSize[i] == 0 &&                                                */
/*        NullRange(initial[i], condition[i], bound[i], step[i])).            */
/* - forall (i = 0; i < nesting; i++) ValidMapping(mapping[i]).               */
/* - forall (i = 0; i < nesting; i++)                                         */
/*       mapping[i] != STHREADS_MAPPING_SIMPLE =>                             */
/*           (numThreads[i] > 0) ||                                           */
/*           (numThreads[i] == 0 &&                                           */
/*            NullRange(initial[i], condition[i], bound[i], step[i])).        */
/* - ValidPriority(priority) || priority == STHREADS_PRIORITY_PARENT.         */
/* - ValidStackSize(stackSize).                                               */
{
    int i;

    CHECKINPUTVALUE(nesting >= 1);
    CHECKINPUTVALUE(chunk != NULL);
    CHECKINPUTVALUE(initial != NULL);
    CHECKINPUTVALUE(condition != NULL);
    for (i = 0; i < nesting; i++)
        CHECKINPUTVALUE(ValidCondition(condition[i]));
    CHECKINPUTVALUE(bound != NULL);
    CHECKINPUTVALUE(step != NULL);
    for (i = 0; i < nesting; i++) {
        if (NullRange(initial[i], condition[i], bound[i], step[i])) break;
        CHECKINPUTVALUE(
            !InfiniteRange(initial[i], condition[i], bound[i], step[i]));
    }
    for (i = 0; i < nesting; i++)
        CHECKINPUTVALUE((chunkSize[i] > 0) ||
                   (chunkSize[i] == 0 &&
                    NullRange(initial[i], condition[i], bound[i], step[i])));
    for (i = 0; i < nesting; i++)
        CHECKINPUTVALUE(ValidMapping(mapping[i]));
    for (i = 0; i < nesting; i++)
        if (mapping[i] != STHREADS_MAPPING_SIMPLE)
            CHECKINPUTVALUE(
                (numThreads[i] > 0) ||
                (numThreads[i] == 0 &&
                 NullRange(initial[i], condition[i], bound[i], step[i])));
    CHECKINPUTVALUE(
        ValidPriority(priority) || priority == STHREADS_PRIORITY_PARENT);
    CHECKINPUTVALUE(ValidStackSize(stackSize));

    return STHREADS_ERROR_NONE;
}

/*----------------------------------------------------------------------------*/
/* Multithreaded general for loop (for future release?)                       */
/*----------------------------------------------------------------------------*/

int SthreadsGeneralForLoop(
        void (*body)(void *control, void *args),
        size_t controlSize, void *args,
        int (*test)(void *args), void (*increment)(void *args),
        void (*copy)(void *control, void *args),
        int mapping, int numThreads,
        int priority, unsigned int stackSize)
/* Arguments:                                                                 */
/* - body        : function to execute one iteration of loop body.            */
/* - controlSize : size (as returned by sizeof) of control variables.         */
/* - args        : pointer to arguments of loop.                              */
/* - test        : function to test loop termination condition.               */
/* - increment   : function to increment control variables within arguments.  */
/* - copy        : function to copy control variables from arguments.         */
/* - mapping     : mapping of iterations onto threads.                        */
/* - numThreads  : number.of threads.                                         */
/* - priority    : priority of threads.                                       */
/* - stackSize   : stack size of threads.                                     */
/* Returns:                                                                   */
/* - error code.                                                              */
/* Requirements:                                                              */
/* - body != NULL &&                                                          */
/*   body is a valid void (*)(void *, void *) function.                       */
/* - test != NULL &&                                                          */
/*   test is a valid int (*)(void *) function.                                 */
/* - increment != NULL &&                                                        */
/*   increment is a valid void (*)(void *) function.                            */
/* - copy != NULL &&                                                             */
/*   copy is a valid void (*)(void *, void *) function.                         */
/* - mapping == STHREADS_MAPPING_SIMPLE ||                                       */
/*   mapping == STHREADS_MAPPING_DYNAMIC.                                        */
/* - mapping != STHREADS_MAPPING_SIMPLE =>                                       */
/*       (numThreads > 0) || (numThreads == 0 && !test(args)).                  */
/* - ValidPriority(priority) || priority == STHREADS_PRIORITY_PARENT.           */
/* - ValidStackSize(stackSize).                                                  */
{
    CHECKINPUTVALUE(body != NULL);
    CHECKINPUTVALUE(test != NULL);
    CHECKINPUTVALUE(increment != NULL);
    CHECKINPUTVALUE(copy != NULL);
    CHECKINPUTVALUE(mapping == STHREADS_MAPPING_SIMPLE ||
                mapping == STHREADS_MAPPING_DYNAMIC);
    if (mapping != STHREADS_MAPPING_SIMPLE)
        CHECKINPUTVALUE((numThreads > 0) || (numThreads == 0 && !test(args)));
    CHECKINPUTVALUE(
        ValidPriority(priority) || priority == STHREADS_PRIORITY_PARENT);
    CHECKINPUTVALUE(ValidStackSize(stackSize));

    return STHREADS_ERROR_NONE;
}

/*--------------------------------------------------------------------------------*/
/* Synchronization object status constants                                        */
/*--------------------------------------------------------------------------------*/

#define INITIALIZED 123456
#define FINALIZED   654321

/*--------------------------------------------------------------------------------*/
/* Flags                                                                          */
/*--------------------------------------------------------------------------------*/

typedef struct {
    int initialized, finalized;
    LONG numWaiting;
    HANDLE signal;
} PrivateFlag;

#define PRIVATE(flagPtr) ((PrivateFlag *) (flagPtr))

/*--------------------------------------------------------------------------------*/

int StreadsFlagInitialize(SthreadsFlag *flag)
{
    CHECKINPUTVALUE(flag != NULL);

    PRIVATE(flag)->initialized = INITIALIZED;
    PRIVATE(flag)->finalized = ~FINALIZED;
    PRIVATE(flag)->numWaiting = 0;
    PRIVATE(flag)->signal = CreateEvent(NULL, TRUE, FALSE, NULL);
    CHECKSYNCCREATE(PRIVATE(flag)->signal != NULL);

    return STHREADS_ERROR_NONE;
}

/*--------------------------------------------------------------------------------*/

int SthreadsFlagFinalize(SthreadsFlag *flag)
{
    BOOL returnOK;

    CHECKINPUTVALUE(flag != NULL);
    CHECKUNINITIALIZED(PRIVATE(flag)->initialized == INITIALIZED);
    CHECKFINALIZED(PRIVATE(flag)->finalized == ~FINALIZED);
    CHECKINUSE(PRIVATE(flag)->numWaiting == 0);

    PRIVATE(flag)->finalized = FINALIZED;
    returnOK = CloseHandle(PRIVATE(flag)->signal);
    CHECKOTHER(returnOK ==       ;

    return STHREADS_ERROR_NONE;
}

/*----------------------------------------------------------------------------*/

int SthreadsFlagSet(SthreadsFlag *flag)
{
    BOOL returnOK;

    CHECKINPUTVALUE(flag != NULL);
    CHECKUNINITIALIZED(PRIVATE(flag)->initialized == INITIALIZED);
    CHECKFINALIZED(PRIVATE(flag)->finalized == ~FINALIZED);

    returnOK = SetEvent(PRIVATE(flag)->signal);
    CHECKOTHER(returnOK);

    return STHREADS_ERROR_NONE;
}

/*----------------------------------------------------------------------------*/

int SthreadsFlagCheck(SthreadsFlag *flag)
{
    DWORD returnCode;

    CHECKINPUTVALUE(flag != NULL);
    CHECKUNINITIALIZED(PRIVATE(flag)->initialized == INITIALIZED);
    CHECKFINALIZED(PRIVATE(flag)->finalized == ~FINALIZED);

    InterlockedIncrement(&PRIVATE(flag)->numWaiting);
    returnCode = WaitForSingleObject(PRIVATE(flag)->signal, INFINITE);
    CHECKOTHER(returnCode != WAIT_FAILED);
    InterlockedDecrement(&PRIVATE(flag)->numWaiting);

    return STHREADS_ERROR_NONE;
}

/*----------------------------------------------------------------------------*/

int SthreadsFlagReset(SthreadsFlag *flag)
{
    BOOL returnOK;

    CHECKINPUTVALUE(flag != NULL);
    CHECKUNINITIALIZED(PRIVATE(flag)->initialized == INITIALIZED);
    CHECKFINALIZED(PRIVATE(flag)->finalized == ~FINALIZED);
    CHECKINUSE(PRIVATE(flag)->numWaiting == 0);

    PRIVATE(flag)->numWaiting = 0;
    returnOK = ResetEvent(PRIVATE(flag)->signal);
    CHECKOTHER(returnOK);

    return STHREADS_ERROR_NONE;
}

/*----------------------------------------------------------------------------*/

#undef PRIVATE

/*----------------------------------------------------------------------------*/
/* Counters                                                                   */
/*----------------------------------------------------------------------------*/

typedef struct node *link;
typedef struct node {
    unsigned int value;
    int numWaiting;
    HANDLE signal;
    link next;
} node;

typedef struct {
    int initialized, finalized;
    unsigned int count;
    link waitingList;
    CRITICAL_SECTION lock;
} PrivateCounter;

#define PRIVATE(counterPtr) ((PrivateCounter *) (counterPtr))

/*-----------------------------------------------------------------------------*/

int SthreadsCounterInitialize(SthreadsCounter *counter)
{
    link startSentinel, endSentinel;

    CHECKINPUTVALUE(counter != NULL);

    PRIVATE(counter)->initialized = INITIALIZED;
    PRIVATE(counter)->finalized = ~FINALIZED;
    PRIVATE(counter)->count = 0;
    startSentinel = (link) malloc(sizeof(node));
    CHECKMEMORYALLOC(startSentinel != NULL);
    endSentinel = (link) malloc(sizeof(node));
    CHECKMEMORYALLOC(endSentinel != NULL);
    startSentinel->signal = NULL;
    startSentinel->next = endSentinel;
    startSentinel->numWaiting = 0;
    endSentinel->signal = NULL;
    endSentinel->next = NULL;
    endSentinel->numWaiting = 0;
    PRIVATE(counter)->waitingList = startSentinel;
    InitializeCriticalSection((LPCRITICAL_SECTION) &PRIVATE(counter)->lock);

    return STHREADS_ERROR_NONE;
}

/*-----------------------------------------------------------------------------*/

int SthreadsCounterFinalize(SthreadsCounter *counter)
{
    link p, next;
    BOOL returnOK;

    CHECKINPUTVALUE(counter != NULL);
    CHECKUNINITIALIZED(PRIVATE(counter)->initialized == INITIALIZED);
    CHECKFINALIZED(PRIVATE(counter)->finalized == ~FINALIZED);
    CHECKINUSE(PRIVATE(counter)->waitingList->next->next == NULL);

    PRIVATE(counter)->finalized = FINALIZED;
    p = PRIVATE(counter)->waitingList;
    next = p->next;
    free(p);
    p = next;
    while (p->next != NULL) {
        returnOK = CloseHandle(p->signal);
        CHECKOTHER(returnOK == TRUE);
        next = p->next;
        free(p);
        p = next;
    }
    free(p);
    DeleteCriticalSection((LPCRITICAL_SECTION) &PRIVATE(counter)->lock);

    return STHREADS_ERROR_NONE;
}

/*-----------------------------------------------------------------------------*/

int SthreadsCounterIncrement(SthreadsCounter *counter, unsigned int amount)
{
    link start, p;
    BOOL returnOK;

    CHECKINPUTVALUE(counter != NULL);
    CHECKUNINITIALIZED(PRIVATE(counter)->initialized == INITIALIZED);
    CHECKFINALIZED(PRIVATE(counter)->finalized == ~FINALIZED);
    CHECKCOUNTEROVERFLOW(PRIVATE(counter)->count <= UINT_MAX - amount);

    EnterCriticalSection((LPCRITICAL_SECTION) &PRIVATE(counter)->lock);
    PRIVATE(counter)->count = PRIVATE(counter)->count + amount;
    start = PRIVATE(counter)->waitingList;
    p = start->next;
    while (p->next != NULL && p->value <= PRIVATE(counter)->count) {
        returnOK = SetEvent(p->signal);
        CHECKOTHER(returnOK);
        start->next = p->next;
        p = start->next;
    }
    LeaveCriticalSection((LPCRITICAL_SECTION) &PRIVATE(counter)->lock);

    return STHREADS_ERROR_NONE;
}

/*------------------------------------------------------------------------------*/

int SthreadsCounterCheck(SthreadsCounter *counter, unsigned int value)
{
    link prev, p;
    link waitingNode;
    BOOL returnOK;
    DWORD returnCode;

    CHECKINPUTVALUE(counter != NULL);
    CHECKUNINITIALIZED(PRIVATE(counter)->initialized == INITIALIZED);
    CHECKFINALIZED(PRIVATE(counter)->finalized == ~FINALIZED);

    EnterCriticalSection((LPCRITICAL_SECTION) &PRIVATE(counter)->lock);
    if (PRIVATE(counter)->count >= value)
        LeaveCriticalSection((LPCRITICAL_SECTION) &PRIVATE(counter)->lock);
    else {
        prev = PRIVATE(counter)->waitingList;
        p = prev->next;
        while (p->next != NULL && p->value < value) {
            prev = p;
            p = p->next;
        }
        if (p->value == value) {
            waitingNode = p;
            waitingNode->numWaiting = waitingNode->numWaiting + 1;
        } else {
            waitingNode = (link) malloc(sizeof(node));
            waitingNode->value = value;
            waitingNode->signal = CreateEvent(NULL, TRUE, FALSE, NULL);
            waitingNode->next = p;
            waitingNode->numWaiting = 1;
            prev->next = waitingNode;
        }
        LeaveCriticalSection((LPCRITICAL_SECTION) &PRIVATE(counter)->lock);
        returnCode = WaitForSingleObject(waitingNode->signal, INFINITE);
        CHECKOTHER(returnCode != WAIT_FAILED);
        EnterCriticalSection((LPCRITICAL_SECTION) &PRIVATE(counter)->lock);
        waitingNode->numWaiting = waitingNode->numWaiting - 1;
        if (waitingNode->numWaiting == 0) {
            returnOK = CloseHandle(waitingNode->signal);
            CHECKOTHER(returnOK == TRUE);
            free(waitingNode);
        }
        LeaveCriticalSection((LPCRITICAL_SECTION) &PRIVATE(counter)->lock);
    }

    return STHREADS_ERROR_NONE;
}

/*------------------------------------------------------------------------------*/

int SthreadsCounterReset(SthreadsCounter *counter)
{
    link p, q;
    BOOL returnOK;
    CHECKINPUTVALUE(counter != NULL);
    CHECKUNINITIALIZED(PRIVATE(counter)->initialized == INITIALIZED);
    CHECKFINALIZED(PRIVATE(counter)->finalized == ~FINALIZED);
    CHECKINUSE(PRIVATE(counter)->waitingList->next->next == NULL);

    PRIVATE(counter)->count = 0;
    p = PRIVATE(counter)->waitingList;
    q = p->next;
    while (q->next != NULL) {
        p->next = q->next;
        returnOK = CloseHandle(q->signal);
        CHECKOTHER(returnOK == TRUE);
        free(q);
        q = p->next;
    }

    return STHREADS_ERROR_NONE;
}

/*----------------------------------------------------------------------------*/

#undef PRIVATE

/*----------------------------------------------------------------------------*/
/* Locks                                                                      */
/*----------------------------------------------------------------------------*/

typedef struct {
    int initialized, finalized;
    HANDLE holder;
    CRITICAL_SECTION lock;
} PrivateLock;

#define PRIVATE(lockPtr) ((PrivateLock *) (lockPtr))

/*----------------------------------------------------------------------------*/

int SthreadsLockInitialize(SthreadsLock *lock)
{
    CHECKINPUTVALUE(lock != NULL);

    PRIVATE(lock)->initialized = INITIALIZED;
    PRIVATE(lock)->finalized = ~FINALIZED;
    PRIVATE(lock)->holder = NULL;
    InitializeCriticalSection((LPCRITICAL_SECTION) &PRIVATE(lock)->lock);

    return STHREADS_ERROR_NONE;
}

/*----------------------------------------------------------------------------*/

int SthreadsLockFinalize(SthreadsLock *lock)
{
    CHECKINPUTVALUE(lock != NULL);
    CHECKUNINITIALIZED(PRIVATE(lock)->initialized == INITIALIZED);
    CHECKFINALIZED(PRIVATE(lock)->finalized == ~FINALIZED);
    CHECKINUSE(PRIVATE(lock)->holder == NULL);

    PRIVATE(lock)->finalized = FINALIZED;
    DeleteCriticalSection((LPCRITICAL_SECTION) &PRIVATE(lock)->lock);

    return STHREADS_ERROR_NONE;
}

/*----------------------------------------------------------------------------*/

int SthreadsLockAcquire(SthreadsLock *lock)
{
    HANDLE thisThread;

    thisThread = GetCurrentThread();
    believe(thisThread!= NULL);
    CHECKINPUTVALUE(lock != NULL);
    CHECKUNINITIALIZED(PRIVATE(lock)->initialized == INITIALIZED);
    CHECKFINALIZED(PRIVATE(lock)->finalized == ~FINALIZED);

    EnterCriticalSection((LPCRITICAL_SECTION) &PRIVATE(lock)->lock);
    believe(PRIVATE(lock)->holder == NULL ||
            PRIVATE(lock)->holder == thisThread);
    CHECKLOCKHELD(PRIVATE(lock)->holder == NULL);
    PRIVATE(lock)->holder = thisThread;

    return STHREADS_ERROR_NONE;
}

/*-----------------------------------------------------------------------------*/

int SthreadsLockRelease(SthreadsLock *lock)
{
    HANDLE thisThread;

    thisThread = GetCurrentThread();
    believe(thisThread!= NULL);

    CHECKINPUTVALUE(lock != NULL);
    CHECKUNINITIALIZED(PRIVATE(lock)->initialized == INITIALIZED);
    CHECKFINALIZED(PRIVATE(lock)->finalized == ~FINALIZED);
    CHECKLOCKNOTHELD(PRIVATE(lock)->holder == thisThread);

    PRIVATE(lock)->holder = NULL;
    LeaveCriticalSection((LPCRITICAL_SECTION) &PRIVATE(lock)->lock);

    return STHREADS_ERROR_NONE;
}

/*-----------------------------------------------------------------------------*/

#undef PRIVATE

/*-----------------------------------------------------------------------------*/
/* Barriers                                                                    */
/*-----------------------------------------------------------------------------*/

typedef struct {
    int initialized, finalized;
    int numThreads;
    int numWaiting;
    HANDLE gate[2];
    int currentGate; /* 0 or 1 */
    CRITICAL_SECTION lock;
} PrivateBarrier;

#define PRIVATE(barrierPtr) ((PrivateBarrier *) (barrierPtr))

/*-----------------------------------------------------------------------------*/

int SthreadsBarrierInitialize(SthreadsBarrier *barrier, int numThreads)
{
    CHECKINPUTVALUE(barrier != NULL);
    CHECKINPUTVALUE(numThreads >= 1);

    PRIVATE(barrier)->initialized = INITIALIZED;
    PRIVATE(barrier)->finalized = ~FINALIZED;
    PRIVATE(barrier)->numThreads = numThreads;
    PRIVATE(barrier)->numWaiting = 0;
    PRIVATE(barrier)->gate[0] = CreateEvent(NULL, TRUE, FALSE, NULL);
    CHECKSYNCCREATE(PRIVATE(barrier)->gate[0] != NULL);
    PRIVATE(barrier)->gate[1] = CreateEvent(NULL, TRUE, TRUE, NULL);
    CHECKSYNCCREATE(PRIVATE(barrier)->gate[1] != NULL);
    PRIVATE(barrier)->currentGate = 0;
    InitializeCriticalSection((LPCRITICAL_SECTION) &PRIVATE(barrier)->lock);

    return STHREADS_ERROR_NONE;
}

/*-----------------------------------------------------------------------------*/
```

```
int SthreadsBarrierFinalize(SthreadsBarrier *barrier)
{
    BOOL returnOK;

    CHECKINPUTVALUE(barrier != NULL);
    CHECKUNINITIALIZED(PRIVATE(barrier)->initialized == INITIALIZED);
    CHECKFINALIZED(PRIVATE(barrier)->finalized == ~FINALIZED);
    CHECKINUSE(PRIVATE(barrier)->numWaiting == 0);

    PRIVATE(barrier)->finalized = FINALIZED;
    returnOK = CloseHandle(PRIVATE(barrier)->gate[0]);
    CHECKOTHER(returnOK == TRUE);
    returnOK = CloseHandle(PRIVATE(barrier)->gate[1]);
    CHECKOTHER(returnOK == TRUE);
    DeleteCriticalSection((LPCRITICAL_SECTION) &PRIVATE(barrier)->lock);

    return STHREADS_ERROR_NONE;
}

/*----------------------------------------------------------------------------*/

int SthreadsBarrierPass(SthreadsBarrier *barrier)
{
    int currentGate, nextGate;
    BOOL returnOK;
    DWORD returnCode;

    CHECKINPUTVALUE(barrier != NULL);
    CHECKUNINITIALIZED(PRIVATE(barrier)->initialized == INITIALIZED);
    CHECKFINALIZED(PRIVATE(barrier)->finalized == ~FINALIZED);

    EnterCriticalSection((LPCRITICAL_SECTION) &PRIVATE(barrier)->lock);
    currentGate = PRIVATE(barrier)->currentGate;
    PRIVATE(barrier)->numWaiting = PRIVATE(barrier)->numWaiting + 1;
    if (PRIVATE(barrier)->numWaiting == PRIVATE(barrier)->numThreads) {
        nextGate = (currentGate + 1)%2;
        returnOK = ResetEvent(PRIVATE(barrier)->gate[nextGate]);
        CHECKOTHER(returnOK);
        PRIVATE(barrier)->numWaiting = 0;
        returnOK = SetEvent(PRIVATE(barrier)->gate[currentGate]);
        CHECKOTHER(returnOK);
        PRIVATE(barrier)->currentGate = nextGate;
        LeaveCriticalSection((LPCRITICAL_SECTION) &PRIVATE(barrier)->lock);
    } else {
        LeaveCriticalSection((LPCRITICAL_SECTION) &PRIVATE(barrier)->lock);
        returnCode = WaitForSingleObject(
            PRIVATE(barrier)->gate[currentGate], INFINITE);
        CHECKOTHER(returnCode != WAIT_FAILED);
    }

    return STHREADS_ERROR_NONE;
}

/*----------------------------------------------------------------------------*/

int SthreadsBarrierReset(SthreadsBarrier *barrier, int numThreads)
{
    BOOL returnOK;

    CHECKINPUTVALUE(barrier != NULL);
    CHECKUNINITIALIZED(PRIVATE(barrier)->initialized == INITIALIZED);
    CHECKFINALIZED(PRIVATE(barrier)->finalized == ~FINALIZED);
    CHECKINUSE(PRIVATE(barrier)->numWaiting == 0);
    CHECKINPUTVALUE(numThreads >= 1);

    PRIVATE(barrier)->numThreads = numThreads;
    PRIVATE(barrier)->numWaiting = 0;
    returnOK = ResetEvent(PRIVATE(barrier)->gate[0]);
    CHECKOTHER(returnOK);
    returnOK = SetEvent(PRIVATE(barrier)->gate[1]);
    CHECKOTHER(returnOK);
    PRIVATE(barrier)->currentGate = 0;
    return STHREADS_ERROR_N
}

/*----------------------------------------------------------------------------*/

#undef PRIVATE

/*----------------------------------------------------------------------------*/
/* Priorities                                                                 */
/*----------------------------------------------------------------------------*/

int SthreadsGetCurrentPriority(int *priority)
{
    HANDLE currentThread;
    int currentPriority;

    CHECKINPUTVALUE(priority != NULL);

    currentThread = GetCurrentThread();
    believe(currentThread != NULL);
    currentPriority = GetThreadPriority(currentThread);
    believe(currentPriority != THREAD_PRIORITY_ERROR_RETURN);

    *priority = currentPriority;

    return STHREADS_ERROR_NONE;
}

/*----------------------------------------------------------------------------*/

int SthreadsSetCurrentPriority(int priority)
{
    HANDLE currentThread;
    BOOL returnOK;

    CHECKINPUTVALUE(ValidPriority(priority));

    currentThread = GetCurrentThread();
    believe(currentThread != NULL);
    returnOK = SetThreadPriority(currentThread, priority);
    believe(returnOK);

    return STHREADS_ERROR_NONE;
}

/*----------------------------------------------------------------------------*/
```